SAT/SMT by Example

Dennis Yurichev

SAT/SMT by Example

Dennis Yurichev

February 23, 2024

Abridged contents

Contents

[1] Satisfiability modulo theories
[2] Boolean satisfiability problem

[3] Linear congruential generator
[4] Pseudorandom number generator
[5] Rivest–Shamir–Adleman cryptosystem

[6] If-Then-Else

[7] Deterministic finite automaton

[8] Also Known As
[9] Printed circuit board

[10] Mutually Orthogonal Latin Square
[11] Greatest Common Divisor
[12] Least Common Multiple

Chapter 1

Introduction

> The SAT problem is evidently a killer app, because it is key to the solution of so many other problems.
>
> Donald Knuth, 1974 ACM Turing Award Recipient, from The Art of Computer Programming, section 7.2.2.2

> The practical solving of SAT is a key technology for computer science in the 21st century
>
> Edmund Clarke, 2007 ACM Turing Award Recipient

> The SAT problem is at the core of arguably the most fundamental question in computer science: What makes a problem hard?
>
> Stephen Cook, 1982 ACM Turing Award Recipient

> Major progresses in logic may come from SAT
>
> Moshe Vardi

1.1 What is this all about?

SAT/SMT solvers can be viewed as solvers of huge systems of equations. The difference is that SMT solvers takes systems in arbitrary format, while SAT solvers are limited to Boolean equations in CNF[1] form.

A lot of real world problems can be represented as problems of solving system of equations.

1.2 Praise

See here: `https://smt.st`.

1.3 As recommended reading at several universities

For a list, see: `https://smt.st`.

1.4 Thanks

Armin Biere has patiently answered to my endless and boring questions.

[1] Conjunctive normal form

Leonardo Mendonça de Moura[2], Nikolaj Bjørner[3], Johannes Altmanninger[4] and Mate Soos[5] have also helped.

Masahiro Sakai[6] has helped with numberlink puzzle: 8.8.

Helped with KLEE: Cristian Cadar[7], Frank Busse[8].

Chad Brewbaker, Evan Wallace, Ben Gardiner, and Wojciech Niedbala have fixed some bugs and made improvements.

Xing Shi Cai, Arseny Nerinovsky, Raphael Wimmer, Andrea Jemmett, Jason Bucata and Tom Chase have found many bugs.

English grammar fixes: Priyanshu Jain, John Winston Garth.

Denis Fokin found a typo.

Martin Nyx Brain has helped with CBMC, etc.

1.5 Disclaimer

This collection is a non-academic reading for *end-users*, i.e., programmers, etc.

The author of these lines is no expert in SAT/SMT, by any means. This is not a book, rather a student's notes. Take it with a grain of salt...

Despite the fact there are many examples for Z3 SMT-solver, the author of these lines is not affiliated with the Z3 team in any way... Nor with any other SAT/SMT teams.

1.6 Python

Majority of code in the book is written in Python 3. Basic understanding is recommended.

1.7 Latest versions

Latest version is always available at `https://smt.st/SAT_SMT_by_example.pdf`.

Russian version has been dropped – it's too hard for me to maintain two versions. Sorry.

New parts are appearing here from time to time, see: `https://smt.st/current_tree/ChangeLog`.

1.8 Proofreaders wanted!

You see how horrible my English is? Please do not hesitate to drop me an email about my mistakes: dennis(@)yurichev.com.

1.9 The source code

Some people find it inconvenient to copy&paste source code from this PDF. You can get the source code here: `https://smt.st/src`.

1.10 Subscribe to my news

For news and updates, subscribe to my blog
(RSS).
Also, subscribe to my mailing list by sending
an empty email to **yurichev+subscribe@googlegroups.com**

[2] https://www.microsoft.com/en-us/research/people/leonardo/

[3] https://www.microsoft.com/en-us/research/people/nbjorner/

[4] https://github.com/krobelus

[5] https://www.msoos.org/

[6] https://twitter.com/masahiro_sakai

[7] https://github.com/ccadar

[8] https://github.com/251

1.11 Is it hype? Yet another fad?

Some people say this is just hype. No, SAT is old and fundamental to CS[9]. The reason for increased interest in it is that computers got faster over the last couple of decades, so there are attempts to solve old problems using SAT/SMT, which were inaccessible in past.

One significant step is GRASP SAT solver (1996) [10], known as CDCL[11], the next is Chaff (2001) [12].

In 1999, a new paper proposed using SAT instead of BDD for symbolic model checking: Armin Biere, Alessandro Cimatti, Edmund Clarke, Yunshan Zhu – Symbolic Model Checking without BDDs [13]. See also: http://fmv.jku.at/biere/talks/Biere-CAV18Award-talk.pdf.

SMT-LIB mailing list was created in 2002 [14].

Also, SAT/SMT are not special or unique. There are adjacent area like ASP: Answer Set Programming. CSP: Constraint Satisfaction Problems. Also, Prolog programming language.

1.12 Backward computation

A nice definition from The Elements of Artificial Intelligence Using Common LISP by Steven L. Tanimoto.

Of course, this is also true for SAT/SMT, CSP, etc...

«Unlike other programming languages, Prolog makes it easy to compute functions “backward.”»

1.13 Is this another programming language?

There are folks who come from Haskell, Coq, other theorem provers. There are folks who come from type theory, etc. And sometimes they have a misconception that this is a PL[15] too.

No, it’s not. It’s just a library/framework/API.

1.14 Hacking/tinkering plan

Where can you start hacking?

From easiest to hardest:

- CBMC examples. CBMC[16] is easy to install. Examples are in pure C, so anybody can hack them without additional training. CBMC exists for Windows.
- KLEE examples. Also in pure C, but KLEE is trickier to install, via docker[17]... No Windows supports, AFAIR.
- Z3Py examples. These are for Z3 Python API. Only Python knowledge is required. Grep for ”from z3 import *” in *.py files, these are for Z3Py.
- SMT-LIB 2.x examples. Lispy language, way less handy than Python, C++, C#, etc... But this language is *distilled*, and not *contaminated* by Python’s (or C++’s) specific features, operator overloading, etc, so there you can see a clear border between SMT API and API’s implementation in specific PL.
- SAT examples. Low-level. Like assembly language – hard to understand, hard to use, but the most efficient way of using SAT solvers.

So grab the source code for my book and start tinkering...

[9] Computer science

[10] https://www.cs.cmu.edu/~emc/15-820A/reading/grasp_iccad96.pdf

[11] Conflict-driven clause learning

[12] http://www.princeton.edu/~chaff/publication/DAC2001v56.pdf

[13] http://fmv.jku.at/papers/BiereCimattiClarkeZhu-TACAS99.pdf

[14] https://cs.nyu.edu/pipermail/smt-lib/2002/

[15] Programming Language

[16] http://www.cprover.org/cbmc/

[17] https://klee.github.io/docker/

Chapter 2

Basics

2.1 One-hot encoding

Throughout this book, we'll often use so-called "one-hot encoding". In short, this is:

Decimal	One-hot
0	00000001
1	00000010
2	00000100
3	00001000
4	00010000
5	00100000
6	01000000
7	10000000

Or in reversed form:

Decimal	One-hot
0	10000000
1	01000000
2	00100000
3	00010000
4	00001000
5	00000100
6	00000010
7	00000001

It has several advantages and disadvantages as well. See also: https://en.wikipedia.org/wiki/One-hot. It's worth noting that one-hot encoding is also called "unitary code" in Russian literature.

2.2 SMT-solvers

2.2.1 School-level system of equations

This is school-level system of equations copy-pasted from Wikipedia [1]:

$$\begin{aligned} 3x + 2y - z &= 1 \\ 2x - 2y + 4z &= -2 \\ -x + \tfrac{1}{2}y - z &= 0 \end{aligned}$$

Will it be possible to solve it using Z3? Here it is:

[1] https://en.wikipedia.org/wiki/System_of_linear_equations

```
#!/usr/bin/python
from z3 import *

x = Real('x')
y = Real('y')
z = Real('z')
s = Solver()
s.add(3*x + 2*y - z == 1)
s.add(2*x - 2*y + 4*z == -2)
s.add(-x + 0.5*y - z == 0)
print s.check()
print s.model()
```

We see this after run:

```
sat
[z = -2, y = -2, x = 1]
```

If we change any equation in some way so it will have no solution, s.check() will return "unsat".

I've used "Real" *sort* (some kind of data type in SMT-solvers) because the last expression equals to $\frac{1}{2}$, which is, of course, a real number. For the integer system of equations, "Int" *sort* would work fine.

Python (and other high-level PLs like C#) interface is highly popular, because it's practical, but in fact, there is a standard language for SMT-solvers called SMT-LIB [2].

Our example rewritten to it looks like this:

```
(declare-const x Real)
(declare-const y Real)
(declare-const z Real)
(assert (=(-(+(* 3 x) (* 2 y)) z) 1))
(assert (=(+(-(* 2 x) (* 2 y)) (* 4 z)) -2))
(assert (=(-(+ (- 0 x) (* 0.5 y)) z) 0))
(check-sat)
(get-model)
```

This language is very close to LISP, but is somewhat hard to read for untrained eyes.

Now we run it:

```
% z3 -smt2 example.smt
sat
(model
  (define-fun z () Real
    (- 2.0))
  (define-fun y () Real
    (- 2.0))
  (define-fun x () Real
    1.0)
)
```

So when you look back to my Python code, you may feel that these 3 expressions could be executed. This is not true: Z3Py API offers overloaded operators, so expressions are constructed and passed into the guts of Z3 without any execution [3]. I would call it "embedded DSL[4]".

Same thing for Z3 C++ API, you may find there "operator+" declarations and many more [5].

Z3 API[6]s for Java, ML and .NET are also exist [7].

Z3Py tutorial: https://github.com/ericpony/z3py-tutorial.

Z3 tutorial which uses SMT-LIB language: http://rise4fun.com/Z3/tutorial/guide.

[2] http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2017-07-18.pdf

[3] https://github.com/Z3Prover/z3/blob/6e852762baf568af2aad1e35019fdf41189e4e12/src/api/python/z3.py

[4] Domain-specific language

[5] https://github.com/Z3Prover/z3/blob/6e852762baf568af2aad1e35019fdf41189e4e12/src/api/c%2B%2B/z3%2B%2B.h

[6] Application programming interface

[7] https://github.com/Z3Prover/z3/tree/6e852762baf568af2aad1e35019fdf41189e4e12/src/api

2.2.2 Another school-level system of equations

I've found this somewhere at Facebook:

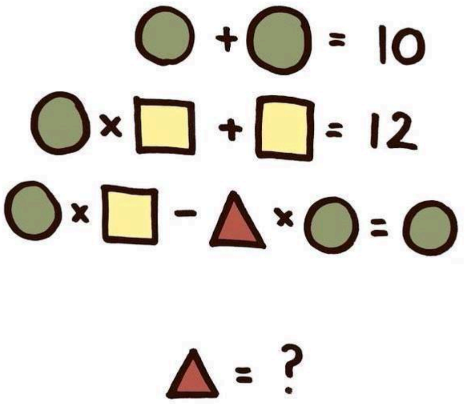


Figure 2.1: System of equations

It's that easy to solve it in Z3:

```
#!/usr/bin/python
from z3 import *

circle, square, triangle = Ints('circle square triangle')
s = Solver()
s.add(circle+circle==10)
s.add(circle*square+square==12)
s.add(circle*square-triangle*circle==circle)
print s.check()
print s.model()
```

```
sat
[triangle = 1, square = 2, circle = 5]
```

2.2.3 Why variables are declared using *declare-fun*?

They mean this is nullary function, only returning a constant, nothing else.

SMT-LIB language has no idea about variables.

Can you get rid of variables? For example, in C/C++ you can use this function for Π constant instead of accessing a constant variable or using preprocessor macro:

```
double PI ()
{
        return 3.1415926;
};
```

In SMT-LIB world, you don't declare variables, you declare argumentless functions that value is either fixed:

```
...
(declare-fun const1 () (_ BitVec 32))
(assert (= const1 (_ bv214013 32)))
...
```

... or it's *connected* to another expression/function's value/etc. This is why variables are declared using `declare-fun` directive. (However, Z3 support `declare-const` directive, which is merely a syntactic sugar.)

One can go further: there are programming languages with no variables, but they are not very practical for our mundane endeavours: https://en.wikipedia.org/wiki/Lambda_calculus.

(And to extreme minimalism: https://en.wikipedia.org/wiki/Iota_and_Jot, https://en.wikipedia.org/wiki/Unlambda.)

LISP has borrowed from some features of lambda calculus, while SMT-LIB language is a very simplified LISP.

2.2.4 Connection between SAT and SMT solvers

SMT-solvers are frontends to SAT solvers, i.e., they translate inputted SMT expressions into CNF and feed SAT-solver with it. Translation process is sometimes called “bit blasting”. Some SMT-solvers uses external SAT-solver: STP uses MiniSAT or CryptoMiniSAT as backend. Some other SMT-solvers (like Z3) has their own SAT solver.

2.2.5 Referential transparency

This is important – a variable can be assigned only once. Like in Verilog/VHDL, Haskell.

Like in SSA[8]:

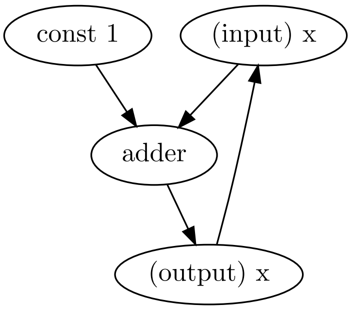
> One important property that holds for all varieties of SSA, including the simplest definition above, is referential transparency: i.e., since there is only a single definition for each variable in the program text, a variable’s value is independent of its position in the program.

(Static Single Assignment Book)

This is like immutability.

Hence, the order of variable assignments doesn’t matter at all. You see, how I do this in my MK85 toy solver: 23.3.1 (`add_FA()` function).

Hence, an expression like $x = x + 1$ wouldn’t work. How can you synthesize it?



You see, such a digital circuit would be senseless.

However, you still can feed this expression to my MK85 toy solver (which is dumb enough to process such an expression) and a circuit like this would be synthesized, but the result would be UNSAT.

Optimization

All this can help in optimization.

For example, in my `SAT_lib.py` Python library I’m using across this book in SAT examples, each gate generated only once for the specific inputs:

```
    def AND(self, v1:int, v2:int) -> int:
        if (v1, v2) in self.AND_already_generated:
            return self.AND_already_generated[(v1,v2)]

        out=self.create_var()
        self.AND_Tseitin(v1, v2, out)

        self.AND_already_generated[(v1,v2)]=out
        return out
```

You can use *cache* or *memoization* words here.

In my toy MK85 SMT solver I’m going further: I cache expressions as strings. Maybe this is a crude hack, but it works:

[8] Static single assignment form

```
struct ctx
{
...
        std::unordered_map<std::string, struct SMT_var*> gen_cache;
};

...

struct SMT_var* gen(struct ctx* ctx, struct expr* e)
{
...
        std::string tmp=cpp_expr_to_string(e);
        auto tmp2=ctx->gen_cache.find(tmp);

        if (tmp2!=ctx->gen_cache.end())
        {
                if (verbose)
                        printf ("found this expression in gen_cache\n");
                return tmp2->second;
        }
        else
        {
                if (verbose)
                        printf ("not found this expression in gen_cache\n");
        };

        ...

                case EXPR_BINARY:
                        if (verbose)
                                printf ("%s:%d (after EXPR_BINARY) end\n",
                                    __FUNCTION__, __LINE__);
                        rt=gen_binary(ctx, e);
                        rt->e=e;
                        ctx->gen_cache[tmp]=rt;
                        return rt;
...
```

This is like *common subexpression elimination*. It would be way harder doing it in Algol-like programming languages.

I added this optimization to MK85 when it failed to generate the *bit reverse* function (4.7) – because the expression *explode* on each iteration drastically.

2.2.6 Theories

QF_S – strings. You may need this to simulate strings to catch bugs like SQL injection. http://cvc4.cs.stanford.edu/wiki/Strings.

```
* QF_UF (Uninterpreted Functions): quantifier-free formulas whose
  satisfiability is to be decided modulo the empty theory. Each
  benchmark may introduce its own uninterpreted function and predicate
  symbols.

* QF_IDL (Integer Difference Logic): quantifier-free formulas to be
  tested for satisfiability modulo a background theory of integer
  arithmetic.  The syntax of atomic formulas is restricted to
  difference logic, i.e. x - y op c, where op is either equality or
  inequality and c is an integer constant.

* QF_RDL (Real Difference Logic): this division is like QF_IDL, except
  that the background theory is real arithmetic.
```

```
* QF_UFIDL (Integer Difference Logic with Uninterpreted Functions):
  this division contains benchmarks in a logic which is similar to
  QF_IDL, except that it also allows uninterpreted functions and
  predicates.

* QF_LIA (Linear Integer Arithmetic): quantifier-free formulas to be
  tested for satisfiability modulo a background theory of integer
  arithmetic.  The syntax of atomic formulas is restricted to contain
  only linear terms.

* QF_LRA (Linear Real Arithmetic): this division is like QF_LIA,
  except that the background theory is real arithmetic.

* QF_NIA (Nonlinear Integer Arithmetic): quantifier-free formulas to be
  tested for satisfiability modulo a background theory of integer
  arithmetic.  There is no restriction to linear terms.

* QF_UFLIA (Linear Integer Arithmetic with Uninterpreted Functions):
  this division contains benchmarks in a logic which is similar to
  QF_LIA, except that it also allows uninterpreted functions and
  predicates.

* QF_UFLRA (Linear Real Arithmetic with Uninterpreted Functions):
  this division is similar to QF_LRA, except that it also allows
  uninterpreted functions and predicates.

* QF_AX (Arrays with Extensionality): quantifier-free formulas to be
  tested for satisfiability modulo a background theory of arrays which
  includes the extensionality axiom.

* QF_AUFLIA (Linear Integer Arithmetic with Uninterpreted Functions
  and Arrays): quantifier-free formulas to be tested for
  satisfiability modulo a background theory combining linear integer
  arithmetic, uninterpreted function and predicate symbols, and
  extensional arrays.

* QF_BV (Fixed-size Bit-vectors): quantifier-free formulas over bit
  vectors of fixed size.

* QF_AUFBV (Bit-vectors with Arrays and Uninterpreted Functions):
  quantifier-free formulas over bit vectors of fixed size, with arrays
  and uninterpreted functions and predicate symbols.

* AUFLIA+p (Linear Integer Arithmetic with Uninterpreted Functions and
  Arrays, with patterns): quantified formulas to be tested for
  satisfiability modulo a background theory combining linear integer
  arithmetic, uninterpreted function and predicate symbols, and
  extensional arrays. Benchmarks include patterns for guiding
  instantiation mechanisms.

* AUFLIA-p (Linear Integer Arithmetic with Uninterpreted Functions and
  Arrays, without patterns): formulas from AUFLIA+p once all patterns
  have been removed.

* AUFLIRA (Arrays, Uninterpreted Functions, and Linear Arithmetic):
  quantifier formulas with arrays of reals indexed by integers
  (Array1), arrays of Array1 indexed by integers (Array2), and linear
  arithmetic over the integers and reals.

* UFNIA+p (Uninterpreted Functions and Nonlinear Arithmetic, patterns
  included): quantifier formulas with uninterpreted functions and
  nonlinear integer arithmetic.

* AUFNIRA (Arrays, Uninterpreted Functions, and Nonlinear Arithmetic):
  quantifier formulas with arrays of reals indexed by integers
  (Array1), arrays of Array1 indexed by integers (Array2), and
  nonlinear arithmetic over the integers and reals.
```

(src)

Listing 2.1: Nicely commented smt_logic_codes.h file from Yices

```
/*
 * This file is part of the Yices SMT Solver.
 * Copyright (C) 2017 SRI International.

...

/*
 * Codes for the logic (based on benchmarks available in June 2009)
 * + one special code 'NONE' for propositional logic
 *
 * 06/03/2014: More surprise logics for SMTCOMP 2014
 *
 * 06/18/2014: Attempt to be a bit more systematic
 * -----------------------------------------------
 * Added all the logics listed at smt-lib.org + a few more that should be there.
 *
 * The base theories are:
 *    AX: arrays
 *    BV: bitvectors
 *    UF: uninterpreted types and functions
 *  + arithmetic
 *
 * For arithmetic:
 * - the domain can be Int or Reals or Both (mixed arithmetic)
 * - the type of atoms can be
 *   difference logic
 *   linear equalities and inequalities
 *   non-linear equalities and inequalities (polynomials)
 * - this gives nine arithmetic variants, but the standard does not include
 *   mixed difference logic. So we have eight arithmetic codes:
 *   IDL, RDL, LIA, LRA, LIRA, NIA, NRA, NIRA
 *
 * AX + UF can be combined with BV and with one of the arithmetic fragments
 * (except that we don't have AIDL and ARDL?).
 *
 * Then, for each logic, we have a quantifier-free variant.
 */
typedef enum smt_logic {
  NONE,         // added 12/27/2012

  /*
   * Base logics (with quantifiers)
   */
  AX,           // arrays
  BV,           // bitvectors
  IDL,          // integer difference logic
  LIA,          // linear integer arithmetic
  LRA,          // linear real arithmetic
  LIRA,         // mixed linear arithmetic
  NIA,          // non-linear integer arithmetic
```

```
NRA,           // non-linear real arithmetic
NIRA,          // non-linear mixed arithmetic
RDL,           // real difference logic
UF,            // uninterpreted functions

//  Arrays + some other theory
ABV,           // arrays + bitvectors
ALIA,          // arrays + linear integer arithmetic
ALRA,          // arrays + linear real arithmetic
ALIRA,         // arrays + mixed linear arithmetic
ANIA,          // arrays + non-linear integer arithmetic
ANRA,          // arrays + non-linear real arithmetic
ANIRA,         // arrays + mixed/non-linear arithmetic
AUF,           // arrays + uninterpreted functions

//  Uninterpreted function + another theory
UFBV,          // uninterpreted functions + bitvectors
UFIDL,         // uninterpreted functions + integer difference logic
UFLIA,         // uninterpreted functions + linear integer arithmetic
UFLRA,         // uninterpreted functions + linear real arithmetic
UFLIRA,        // uninterpreted functions + mixed linear arithmetic
UFNIA,         // uninterpreted functions + non-linear integer arithmetic
UFNRA,         // uninterpreted functions + non-linear real arithmetic
UFNIRA,        // uninterpreted functions + mixed, non-linear arithmetic
UFRDL,         // uninterpreted functions + real difference logic

//  Arrays + uninterpreted functions + another theory
AUFBV,         // arrays + uninterpreted functions + bitvectors
AUFLIA,        // arrays + uninterpreted functions + linear integer arithmetic
AUFLRA,        // arrays + uninterpreted functions + linear real arithmetic
AUFLIRA,       // arrays + uninterpreted functions + mixed linear arithmetic
AUFNIA,        // arrays + uninterpreted functions + non-linear integer arithmetic
AUFNRA,        // arrays + uninterpreted functions + non-linear real arithmetic
AUFNIRA,       // arrays + uninterpreted functions + mixed, non-linear arithmetic

/*
 * Quantifier-free fragments
 */
QF_AX,         // arrays
QF_BV,         // bitvectors
QF_IDL,        // integer difference logic
QF_LIA,        // linear integer arithmetic
QF_LRA,        // linear real arithmetic
QF_LIRA,       // mixed linear arithmetic
QF_NIA,        // non-linear integer arithmetic
QF_NRA,        // non-linear real arithmetic
QF_NIRA,       // non-linear mixed arithmetic
QF_RDL,        // real difference logic
QF_UF,         // uninterpreted functions

//  Arrays + some other theory
QF_ABV,        // arrays + bitvectors
QF_ALIA,       // arrays + linear integer arithmetic
QF_ALRA,       // arrays + linear real arithmetic
QF_ALIRA,      // arrays + mixed linear arithmetic
QF_ANIA,       // arrays + non-linear integer arithmetic
QF_ANRA,       // arrays + non-linear real arithmetic
QF_ANIRA,      // arrays + mixed/non-linear arithmetic
QF_AUF,        // arrays + uninterpreted functions

//  Uninterpreted function + another theory
```

```
  QF_UFBV,     // uninterpreted functions + bitvectors
  QF_UFIDL,    // uninterpreted functions + integer difference logic
  QF_UFLIA,    // uninterpreted functions + linear integer arithmetic
  QF_UFLRA,    // uninterpreted functions + linear real arithmetic
  QF_UFLIRA,   // uninterpreted functions + mixed linear arithmetic
  QF_UFNIA,    // uninterpreted functions + non-linear integer arithmetic
  QF_UFNRA,    // uninterpreted functions + non-linear real arithmetic
  QF_UFNIRA,   // uninterpreted functions + mixed, non-linear arithmetic
  QF_UFRDL,    // uninterpreted functions + real difference logic

  //  Arrays + uninterpreted functions + another theory
  QF_AUFBV,    // arrays + uninterpreted functions + bitvectors
  QF_AUFLIA,   // arrays + uninterpreted functions + linear integer arithmetic
  QF_AUFLRA,   // arrays + uninterpreted functions + linear real arithmetic
  QF_AUFLIRA,  // arrays + uninterpreted functions + mixed linear arithmetic
  QF_AUFNIA,   // arrays + uninterpreted functions + non-linear integer arithmetic
  QF_AUFNRA,   // arrays + uninterpreted functions + non-linear real arithmetic
  QF_AUFNIRA,  // arrays + uninterpreted functions + mixed, non-linear arithmetic

...

/*
 * Arithmetic fragments
 */
typedef enum arith_fragment {
  ARITH_IDL,   // integer difference logic
  ARITH_RDL,   // real difference logic
  ARITH_LIA,   // linear integer arithmetic
  ARITH_LRA,   // linear real arithmetic
  ARITH_LIRA,  // linear mixed arithmetic
  ARITH_NIA,   // non-linear integer arithmetic
  ARITH_NRA,   // non-linear real arithmetic
  ARITH_NIRA,  // non-linear mixed arithmetic
  ARITH_NONE,  // no arithmetic
} arith_fragment_t;
```

What is *theory*?

One can say that a SMT solver is a library of various methods on top of a SAT solver. And sometimes, one *theory*/method can perform better for your problem than another.

You can first start without setting theory explicitly or setting QF_ALL. But then you can experiment trying some of them.

It's recommended to set a *theory* that has only features you need, nothing else.

And of course, SMT benchmarks for SMT solvers competition are divided by theories: http://smt-lib.loria.fr/zip.

2.2.7 Modulo *something*

There is an idiom: *modulo errors*.

For example:

> More recently, I learned another useful term: ”modulo errors.” This is used, for example, as ”Q.E.D., modulo errors.” One fellow often applied this to his blackboard proofs, meaning: ”This is a representative of an equivalence class of proofs, one of which is correct and all of which look sort of like this one. At least one such proof is correct, but it might not be the one I wrote down.”

(https://jcdverha.home.xs4all.nl/scijokes/1_7.html)

Another example:

> Exact inference aims at solving the tasks in an exact way, modulo errors of computer floating point arithmetic.

(Foundations of Probabilistic Logic Programming: Languages, Semantics ... — Fabrizio Riguzzi `https://books.google.com/books?id=zk-MDwAAQBAJ`)

Donald Knuth uses this:

> But if anybody has a significantly better idea, I'm happy to listen — modulo the situation discussed in [don't send me email].

(`https://www-cs-faculty.stanford.edu/~knuth/news98.html`)

Also, there is a tool: "MCMT: Model Checker Modulo Theories" (`http://users.mat.unimi.it/users/ghilardi/mcmt/`).

Now what about "satisfiability modulo theories"?

Some thinks that means "satisfiability modulo theories [that are supported by solver]" or *limited/adjusted/augmented* by them.

> According to one dictionary, "modulo" means "correcting or adjusting for something." This is an appropriate definition for its use in satisfiability modulo theories:
>
> First-order logic defines a notion of satisfiability for arbitrary formulas. In SMT, we use the same definition of satisfiability, but *adjusted* by restricting the set of models we consider to those that also satisfy the theories involved.

(Clark Barrett, `https://groups.google.com/u/1/g/smt-lib/c/Jio8V_vc3kU`)

2.2.8 Division by 0

How SMT-solvers handles division by zero? They can't throw exception, like a CPU.

It was decided to *totalize* division operation, i.e., make it producing an output for each input. But which output should be for $\frac{x}{0}$?

One idea was that $\frac{x}{0}$ should be 0, as this is done in theorem-provers.

Another idea is to fix $\frac{x}{0}$ to `0b11...11` or `0xff..ff`:

```
From an implementation point of view, the simplest approach is one that
avoids special treatments for division by zero and makes things uniform.

The majority of implementers have chosen (bvudiv x 0) = 11...1 because
that's what you get from a long division algorithm or divider circuit
based on long division, and I guess that's how most of us implement division
in SMT solvers.

It is also reasonable to define:

     (bvurem x y) = x - (bvudiv x y) * y.

This equality is already required by SMT-LIB when y>0. If we require
the equality to also hold for y=0, we have  (bvurem x 0) = x no matter
how we define (bvudiv x 0). This is again consistent with what the
long-division algorithm produces.
```

(src)

This is quite fun, I tried to divide by zero using my toy-blaster MK85, this resulted in `0xff..ff`, since my solver implements a primitive divisor.

See also, start of discussion.

From the SMT-LIB standard:

```
   [[(bvudiv s t)]] := if bv2nat([[t]]) = 0
                       then λx:[0, m). 1
                       else nat2bv[m](bv2nat([[s]]) div bv2nat([[t]]))



...

 "After extensive discussion, it was decided to fix the value of
   (bvudiv s t) and (bvurem s t) in the case when bv2nat([[t]]) is 0.
   While this solution is not preferred by all users, it has the
   advantage that it simplifies solver implementations.  Furthermore,
   it is straightforward for users to use alternative semantics by
   defining their own version of these operators (using define-fun) and
   using ite to insert their own semantics when the second operand is
   0.
 "
```

If you want to *disable* division by zero, you can still add a constraint preventing divisor to be zero. And if your equation would *require* to divide by 0, you'll get *unsat*.

Anyway, you can always add an ITE, that will branch depending on the value of divisor.

2.2.9 List of SMT-solvers

- Yices[9], created by Bruno Dutertre et al. Logics Yices 2.6.2 supports[10]:

  ALL, BV, LIA, LRA, QF_ABV, QF_ALIA, QF_ALIRA, QF_AUFBV, QF_AUFLIA, QF_AUFLIRA, QF_BV, QF_IDL, QF_LIA, QF_LIRA, QF_LRA, QF_NIA, QF_NIRA, QF_NRA, QF_RDL, QF_UF, QF_UFBV, QF_UFIDL, QF_UFLIA, QF_UFLRA, QF_UFNIA, QF_UFNRA, UF.

- Z3[11], developed by Leonardo de Moura, Nikolaj Bjorner, Christoph M. Wintersteiger, Lev Nachmanson. Has its own state-of-art SAT solver.

  Many examples here uses Python 2.x API for Z3 (AKA Z3Py). Installation instructions (Ubuntu):

  ```
  sudo apt-get install python3-pip
  sudo pip3 install z3-solver
  ```

  Or compile the latest on Ubuntu:

  ```
  git clone https://github.com/Z3Prover/z3.git
  cd z3
  git tag
  git checkout z3-4.12.0  # or newer version
  python3 scripts/mk_make.py --python
  cd build
  make -j8
  sudo make install
  ```

  Python modules goes to `/usr/lib/python3.x/site-packages/z3`. If you have trouble with this, you can install Python modules to your `$HOME`:

  ```
  python3 scripts/mk_make.py --prefix=$HOME --python --pypkgdir=$HOME/.local/lib/
      python3.8/site-packages
  ```

  (Unofficial) bindings: Haskell[12], Racket[13], Rust[14].

  Fun story: SMT Solving on an iPhone.

[9] http://yices.csl.sri.com/

[10] Grep'ed from its test suite

[11] https://github.com/Z3Prover/z3

[12] http://hackage.haskell.org/package/z3, http://leventerkok.github.io/sbv/

[13] https://github.com/philnguyen/z3-rkt, https://github.com/sunshowers/z3.rkt

[14] https://github.com/prove-rs/z3.rs

  The first version of solver was called 'Zapatho', the second 'Zap2', the third 'Z3'[15] Go figure how the fourth and fifth versions would be called.

  Logics Z3 4.8.x supports[16]:

  ABV, ALL, AUFBV, AUFLIA, AUFNIRA, BV, HORN, LIA, NRA, QF_AUFLIA, QF_AUFLIRA, QF_BV, QF_BVFP, QF_FD, QF_FP, QF_FP, QF_FPA, QF_FPBV, QF_IDL, QF_LIA, QF_LRA, QF_NIA, QF_NRA, QF_NRA, QF_RDL, QF_S, QF_UF, QF_UFBV, QF_UFLIA, UFBV, UFLIA.

- STP[17], used in KLEE.

  Logics STP 2.3.3 supports[18]: QF_ABV, QF_AUFBV, QF_BV.

- CVC3/CVC4[19].

  Newest version is CVC5: https://github.com/cvc5/cvc5.

  Logics CVC 1.9 supports[20]:

  ABV, ALL, ALL_SUPPORTED, ASLIA, AUFBV, AUFBVDTLIRA, AUFBVDTNIRA, AUFDTLIA, AUFLIA, AUFLIAFS, AUFLIRA, AUFNIA, AUFNIRA, BV, FP, LIA, LIRA, LRA, NIA, NIRA, NIRAT, NRA, QF_ABV, QF_ALIA, QF_ALL, QF_ALL_SUPPORTED, QF_ANIA, QF_AUF, QF_AUFBV, QF_AUFBVLIA, QF_AUFBVLF QF_AUFLIA, QF_AUFLIRA, QF_AUFNIA, QF_AUFNIRA, QF_AUFNRA, QF_AX, QF_BV, QF_BVDTLIAFS, QF_BVFP, QF_BVFS, QF_BVLIA, QF_DT, QF_DTLIA, QF_FP, QF_FPLRA, QF_IDL, QF_LIA, QF_LIRA, QF_LIRAFS, QF_LRA, QF_NIA, QF_NIRA, QF_NRA, QF_NRAT, QF_RDL, QF_S, QF_SAT, QF_SEP_LIA, QF_SLIA, QF_UF, QF_UFBV, QF_UFBVFS, QF_UFC, QF_UFDTLIA, QF_UFIDL, QF_UFLIA, QF_UFLIAFS, QF_UFLIRA, QF_UFLIRAFS, QF_UFLRA, QF_UFLRAFS, QF_UFNIA, QF_UFNIRA, QF_UFNRA, QF_UFNRAT, QF_UFSLIA, SAT, SLIA, UF, UFBV, UFC, UFDT, UFDTLIA, UFDTLIRA, UFDTSLIA, UFFS, UFIDL, UFLIA, UFLIRA, UFNIA, UFNIRA, UFNRAT.

- Boolector[21], developed by Aina Niemetz, Mathias Preiner and Armin Biere. Known as the fastest bitvector solver.

  Logics Boolector 3.2.1 supports: BV, QF_ABV, QF_AUFBV, QF_BV, QF_UFBV.

- Alt-Ergo[22], used in Frama-C.

- MathSAT[23]. Developed by Alberto Griggio, Alessandro Cimatti and Roberto Sebastiani.

  Logics MathSAT 5.6.5 supports:

  QF_ABV, QF_ALIA, QF_AUFBV, QF_AUFLIA, QF_AX, QF_BV, QF_BVFP, QF_FP, QF_FPBV, QF_LIA, QF_LRA, QF_UF, QF_UFBV, QF_UFLIA, QF_UFLRA.

- veriT[24]. Developed by David Déharbe, Pascal Fontaine, Haniel Barbosa. Lacks bitvectors.

- toysolver[25] by Masahiro Sakai, written in Haskell.

- MK85[26]. Created by the author, as a toy bit-blaster, supports only booleans and bitvectors.

- dReal: "An SMT Solver for Nonlinear Theories of the Reals" [27].

Something else:

- PySMT: unified Python interface to many SMT solvers [28].

- JavaSMT – Unified Java API for SMT solvers: https://github.com/sosy-lab/java-smt

- jSMTLIB – Another Java API for SMT solvers: http://smtlib.github.io/jSMTLIB/

- SBV: SMT Based Verification in Haskell: http://leventerkok.github.io/sbv/

[15] https://github.com/Z3Prover/z3/issues/2321, https://stackoverflow.com/questions/49963012/what-is-the-meaning-of-the-name-of-the-z

[16] Grep'ed from its test suite

[17] https://github.com/stp/stp

[18] Grep'ed from its test suite

[19] http://cvc4.stanford.edu/ AKA *A Cooperating Validity Checker*, Also available online: https://cvc4.github.io/app/, See also: http://cvc4.cs.stanford.edu/wiki/.

[20] Grep'ed from its test suite

[21] http://fmv.jku.at/boolector/

[22] https://alt-ergo.ocamlpro.com/

[23] http://mathsat.fbk.eu/

[24] http://www.verit-solver.org/

[25] https://github.com/msakai/toysolver

[26] https://smt.st/MK85/

[27] http://dreal.cs.cmu.edu, https://github.com/dreal

[28] https://pysmt.readthedocs.io/en/latest/, https://github.com/pysmt/pysmt, http://smt2015.csl.sri.com/wp-content/uploads/2015/06/2015-Gario-Micheli-PySMT-a-Solver-Agnostic-Library-for-Fast-Prototyping-of-SMT-Based-Algorithms.pdf http://fmv.jku.at/avm15/slides/gario.pdf

Boolector

Boolector can generate CNF file instead of handing it out to SAT solver:

```
boolector --smt2 --dump-dimacs filename.smt > filename.cnf
```

Just keep in mind that if Boolector calls a SAT solver several times, all these CNFs would be dumped to stdout, like:

```
c CNF dump 1 start
c Boolector version
p cnf 13325 36769
1 0
2 0

...

-13325 0
c CNF dump 1 end
c CNF dump 2 start
c Boolector version
p cnf 15421 40065
1 0
2 0

...

15421 0
c CNF dump 2 end
c CNF dump 3 start
c Boolector version
p cnf 15488 40267
1 0
2 0
...
```

This is useful: you can try other SAT solvers, proof checkers, etc...

2.2.10 Z3 specific

The output is not guaranteed to be random. You can randomize it by:

```
import time

...

s=Solver()
set_param("smt.random_seed", int(time.time()))
```

Or conversely, you may want to reproduce its result each time the same:

```
set_param("smt.random_seed", 1234)
```

2.3 SAT-solvers

SMT vs. SAT is like high level PL vs. assembly language. The latter can be much more efficient, but it's hard to program in it.

SAT is abbreviation of "Boolean satisfiability problem". The problem is to find such a set of variables, which, if plugged into boolean expression, will result in "true".

2.3.1 CNF form

CNF[29] is a *normal form*.

Any Boolean expression can be converted to *normal form* and CNF is one of them. The CNF expression is a bunch of clauses (sub-expressions) consisting of terms (variables), ORs and NOTs, all of which are then glued together with AND into a full expression. There is a way to memorize it: CNF is "AND of ORs" (or "product of sums") and DNF[30] is "OR of ANDs" (or "sum of products").

Example is: $(\neg A \vee B) \wedge (C \vee \neg D)$.

$\vee$ stands for OR (logical disjunction[31]), "+" sign is also sometimes used for OR.

$\wedge$ stands for AND (logical conjunction[32]). It is easy to memorize: $\wedge$ looks like "A" letter. "·" is also sometimes used for AND.

$\neg$ is negation (NOT).

2.3.2 Example: 2-bit adder

SAT-solver is merely a solver of huge Boolean equations in CNF form. It just gives the answer, if there is a set of input values which can satisfy CNF expression, and what input values must be.

Here is a 2-bit adder for example:

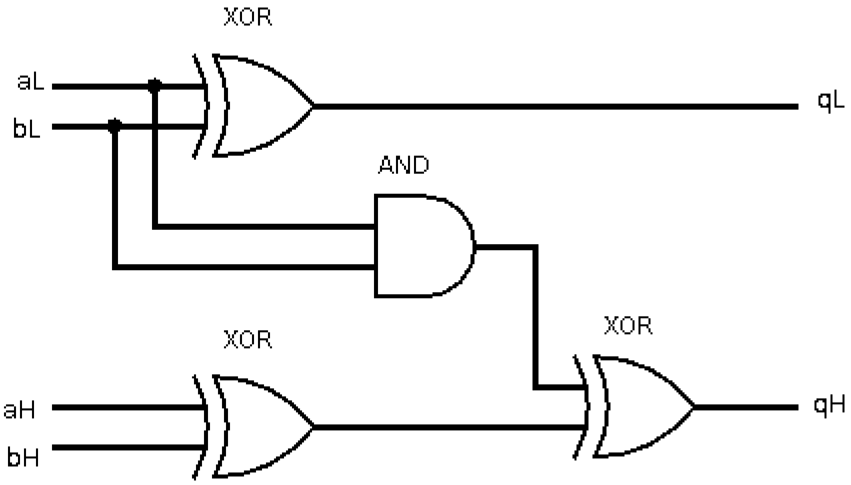


Figure 2.2: 2-bit adder circuit

The adder in its simplest form: it has no carry-in and carry-out, and it has 3 XOR gates and one AND gate. Let's try to figure out, which sets of input values will force adder to set both two output bits? By doing quick memory calculation, we can see that there are 4 ways to do so: $0+3=3$, $1+2=3$, $2+1=3$, $3+0=3$. Here is also truth table, with these rows highlighted:

[29] https://en.wikipedia.org/wiki/Conjunctive_normal_form
[30] Disjunctive normal form
[31] https://en.wikipedia.org/wiki/Logical_disjunction
[32] https://en.wikipedia.org/wiki/Logical_conjunction

	aH	aL	bH	bL	qH	qL
$3+3 = 6 \equiv 2 \pmod 4$	1	1	1	1	1	0
$3+2 = 5 \equiv 1 \pmod 4$	1	1	1	0	0	1
$3+1 = 4 \equiv 0 \pmod 4$	1	1	0	1	0	0
$3+0 = 3 \equiv 3 \pmod 4$	1	1	0	0	1	1
$2+3 = 5 \equiv 1 \pmod 4$	1	0	1	1	0	1
$2+2 = 4 \equiv 0 \pmod 4$	1	0	1	0	0	0
$2+1 = 3 \equiv 3 \pmod 4$	1	0	0	1	1	1
$2+0 = 2 \equiv 2 \pmod 4$	1	0	0	0	1	0
$1+3 = 4 \equiv 0 \pmod 4$	0	1	1	1	0	0
$1+2 = 3 \equiv 3 \pmod 4$	0	1	1	0	1	1
$1+1 = 2 \equiv 2 \pmod 4$	0	1	0	1	1	0
$1+0 = 1 \equiv 1 \pmod 4$	0	1	0	0	0	1
$0+3 = 3 \equiv 3 \pmod 4$	0	0	1	1	1	1
$0+2 = 2 \equiv 2 \pmod 4$	0	0	1	0	1	0
$0+1 = 1 \equiv 1 \pmod 4$	0	0	0	1	0	1
$0+0 = 0 \equiv 0 \pmod 4$	0	0	0	0	0	0

Let's find, what SAT-solver can say about it?

First, we should represent our 2-bit adder as CNF expression.

Using Wolfram Mathematica, we can express 1-bit expression for both adder outputs:

```
In[]:=AdderQ0[aL_,bL_]=Xor[aL,bL]
Out[]:=aL ⊻ bL

In[]:=AdderQ1[aL_,aH_,bL_,bH_]=Xor[And[aL,bL],Xor[aH,bH]]
Out[]:=aH ⊻ bH ⊻ (aL && bL)
```

We need such expression, where both parts will generate 1's. Let's use Wolfram Mathematica find all instances of such expression (I glued both parts with And):

```
In[]:=Boole[SatisfiabilityInstances[And[AdderQ0[aL,bL],AdderQ1[aL,aH,bL,bH]],{aL,aH,bL,bH},4]]
Out[]:={1,1,0,0},{1,0,0,1},{0,1,1,0},{0,0,1,1}
```

Yes, indeed, Mathematica says, there are 4 inputs which will lead to the result we need. So, Mathematica can also be used as SAT solver.

Nevertheless, let's proceed to CNF form. Using Mathematica again, let's convert our expression to CNF form:

```
In[]:=cnf=BooleanConvert[And[AdderQ0[aL,bL],AdderQ1[aL,aH,bL,bH]],``CNF'']
Out[]:=(!aH || !bH) && (aH || bH) && (!aL || !bL) && (aL || bL)
```

Looks more complex. The reason of such verbosity is that CNF form doesn't allow XOR operations.

MiniSat

For starters, we can try MiniSat[33]. The standard way to encode CNF expression for MiniSat is to enumerate all OR parts at each line. Also, MiniSat doesn't support variable names, just numbers. Let's enumerate our variables: 1 will be aH, 2 – aL, 3 – bH, 4 – bL.

[33] http://minisat.se/MiniSat.html

Here is what I've got when I converted Mathematica expression to the MiniSat input file:

```
p cnf 4 4
-1 -3 0
1 3 0
-2 -4 0
2 4 0
```

Two 4's at the first lines are number of variables and number of clauses respectively. There are 4 lines then, each for each OR clause. Minus before variable number meaning that the variable is negated. Absence of minus – not negated. Zero at the end is just terminating zero, meaning end of the clause.

In other words, each line is OR-clause with optional negations, and the task of MiniSat is to find such set of input, which can satisfy all lines in the input file.

That file I named as *adder.cnf* and now let's try MiniSat:

```
% minisat -verb=0 adder.cnf results.txt
SATISFIABLE
```

The results are in *results.txt* file:

```
SAT
-1 -2 3 4 0
```

This means, if the first two variables (aH and aL) will be *false*, and the last two variables (bH and bL) will be set to *true*, the whole CNF expression is satisfiable. Seems to be true: if bH and bL are the only inputs set to *true*, both resulting bits are also has *true* states.

Now how to get other instances? SAT-solvers, like SMT solvers, produce only one solution (or *instance*).

MiniSat uses PRNG and its initial seed can be set explicitly. I tried different values, but result is still the same. Nevertheless, CryptoMiniSat in this case was able to show all possible 4 instances, in chaotic order, though. So this is not very robust way.

Perhaps, the only known way is to negate solution clause and add it to the input expression. We've got `-1 -2 3 4`, now we can negate all values in it (just toggle minuses: `1 2 -3 -4`) and add it to the end of the input file:

```
p cnf 4 5
-1 -3 0
1 3 0
-2 -4 0
2 4 0
1 2 -3 -4
```

Now we've got another result:

```
SAT
1 2 -3 -4 0
```

This means, aH and aL must be both *true* and bH and bL must be *false*, to satisfy the input expression. Let's negate this clause and add it again:

```
p cnf 4 6
-1 -3 0
1 3 0
-2 -4 0
2 4 0
1 2 -3 -4
-1 -2 3 4 0
```

The result is:

```
SAT
-1 2 3 -4 0
```

aH=false, aL=true, bH=true, bL=false. This is also correct, according to our truth table.

Let's add it again:

```
p cnf 4 7
-1 -3 0
```

```
1 3 0
-2 -4 0
2 4 0
1 2 -3 -4
-1 -2 3 4 0
1 -2 -3 4 0
```

```
SAT
1 -2 -3 4 0
```

$aH=true$, $aL=false$, $bH=false$, $bL=true$. This is also correct.

This is fourth result. There are shouldn't be more. What if to add it?

```
p cnf 4 8
-1 -3 0
1 3 0
-2 -4 0
2 4 0
1 2 -3 -4
-1 -2 3 4 0
1 -2 -3 4 0
-1 2 3 -4 0
```

Now MiniSat just says "UNSATISFIABLE" without any additional information in the resulting file.

Our example is tiny, but MiniSat can work with huge CNF expressions.

CryptoMiniSat

XOR operation is absent in CNF form, but crucial in cryptographical algorithms. Simplest possible way to represent single XOR operation in CNF form is: $(\neg x \vee \neg y) \wedge (x \vee y)$ – not that small expression, though, many XOR operations in single expression can be optimized better.

One significant difference between MiniSat and CryptoMiniSat is that the latter supports clauses with XOR operations instead of ORs, because CryptoMiniSat has aim to analyze crypto algorithms[34]. XOR clauses are handled by CryptoMiniSat in a special way without translating to OR clauses.

You need just to prepend a clause with "x" in CNF file and OR clause is then treated as XOR clause by CryptoMiniSat. As of 2-bit adder, this smallest possible XOR-CNF expression can be used to find all inputs where both output adder bits are set:

$(aH \oplus bH) \wedge (aL \oplus bL)$

This is `.cnf` file for CryptoMiniSat:

```
p cnf 4 2
x1 3 0
x2 4 0
```

Now I run CryptoMiniSat with various random values to initialize its PRNG ...

```
% cryptominisat4 --verb 0 --random 0 XOR_adder.cnf
s SATISFIABLE
v 1 2 -3 -4 0
% cryptominisat4 --verb 0 --random 1 XOR_adder.cnf
s SATISFIABLE
v -1 -2 3 4 0
% cryptominisat4 --verb 0 --random 2 XOR_adder.cnf
s SATISFIABLE
v 1 -2 -3 4 0
% cryptominisat4 --verb 0 --random 3 XOR_adder.cnf
s SATISFIABLE
v 1 2 -3 -4 0
% cryptominisat4 --verb 0 --random 4 XOR_adder.cnf
s SATISFIABLE
v -1 2 3 -4 0
% cryptominisat4 --verb 0 --random 5 XOR_adder.cnf
```

[34] http://www.msoos.org/xor-clauses/

```
s SATISFIABLE
v -1 2 3 -4 0
% cryptominisat4 --verb 0 --random 6 XOR_adder.cnf
s SATISFIABLE
v -1 -2 3 4 0
% cryptominisat4 --verb 0 --random 7 XOR_adder.cnf
s SATISFIABLE
v 1 -2 -3 4 0
% cryptominisat4 --verb 0 --random 8 XOR_adder.cnf
s SATISFIABLE
v 1 2 -3 -4 0
% cryptominisat4 --verb 0 --random 9 XOR_adder.cnf
s SATISFIABLE
v 1 2 -3 -4 0
```

Nevertheless, all 4 possible solutions are:

```
v -1 -2 3 4 0
v -1 2 3 -4 0
v 1 -2 -3 4 0
v 1 2 -3 -4 0
```

...the same as reported by MiniSat.

2.3.3 Picosat

At least Picosat can enumerate all possible solutions without crutches I just shown:

```
% picosat --all adder.cnf
s SATISFIABLE
v -1 -2 3 4 0
s SATISFIABLE
v -1 2 3 -4 0
s SATISFIABLE
v 1 2 -3 -4 0
s SATISFIABLE
v 1 -2 -3 4 0
s SOLUTIONS 4
```

2.3.4 MaxSAT

MaxSAT problem is a problem where as many clauses should be satisfied, as possible, but maybe not all.

(Usual) clauses which *must* be satisfied, called *hard clauses.* Clauses which *should* be satisfied, called *soft clauses.*

MaxSAT solver tries to satisfy all *hard clauses* and as much *soft clauses*, as possible.

*.wcnf files are used, the format is almost the same as in DIMACS files, like:

```
p wcnf 207 796 208
208 1 0
208 2 0
208 3 0
208 4 0

...

1 -152 0
1 -153 0
1 -154 0
1 155 0
1 -156 0
1 -157 0
```

Each clause is written as in DIMACS file, but the first number if weight. MaxSAT solver tries to maximize clauses with bigger weights first.

If the weight has *top weight*, the clause is *hard clause* and must always be satisfied. *Top weight* is set in header. In our case, it's 208.

Some well-known MaxSAT solvers are Open-WBO[35], etc.

2.3.5 List of SAT-solvers

- MiniSat[36] by Niklas Een and Niklas Sörensson, serving as a base for some others. `minisat` Ubuntu package.
- PicoSat, PrecoSat, Lingeling, CaDiCaL[37]. All created by Armin Biere. Plingeling supports multithreading. `picosat` Ubuntu package is available.
- CryptoMiniSat[38]. Created by Mate Soos for cryptographical problems exploration. Supports XOR clauses, multithreading. Has Python API.

  See also: Mate Soos, Karsten Nohl, Claude Castelluccia – Extending SAT Solvers to Cryptographic Problems (LNCS, volume 5584) [39].
- The Glucose SAT Solver, based on Minisat[40].
- gophersat, a SAT solver in Go[41].
- microsat[42] by Marijn Heule, smallest known CDCL solver, (238 SLOC of pure C).

MaxSAT solvers:

- Open-WBO[43], by Ruben Martins, Vasco Manquinho, Inês Lynce.

Something else:

- http://www.satcompetition.org/ — benchmarks, etc.
- PySAT: unified interface to many SAT solvers, in Python[44].

Lingeling by Armin Biere

http://fmv.jku.at/lingeling/.

... can be also used as a preprocessor. For example, this will remove tautological clauses:

```
lingeling -s fname.cnf -o - --clim=0
```

What are tautological clauses? These are clauses like "... X ... -X ..." — they can be satisfied by either X=false or X=true, so such a clause doesn't affect anything at all and can be removed or ignored.

Donald Knuth's SAT solvers

For his chapter 7.2.2.2 about SAT solvers 26.2, D.Knuth wrote several SAT solvers:

Listing 2.2: From Donald Knuth's homepage

```
SAT0
    My implementation of Algorithm 7.2.2.2A (very basic SAT solver)
SAT0W
    My implementation of Algorithm 7.2.2.2B (teeny tiny SAT solver)
SAT8
    My implementation of Algorithm 7.2.2.2W (WalkSAT)
SAT9
    My implementation of Algorithm 7.2.2.2S (survey propagation SAT solver)
```

[35] http://sat.inesc-id.pt/open-wbo

[36] http://minisat.se/

[37] https://github.com/arminbiere/cadical

[38] https://www.msoos.org/wordpress/wp-content/uploads/2010/08/cryptominisat-2.5.1.pdf https://github.com/msoos/cryptominisat

[39] https://doi.org/10.1007/978-3-642-02777-2_24

[40] http://www.labri.fr/perso/lsimon/glucose/

[41] https://github.com/crillab/gophersat

[42] https://github.com/marijnheule/microsat/

[43] http://sat.inesc-id.pt/open-wbo/

[44] https://github.com/pysathq/pysat

```
SAT10
    My implementation of Algorithm 7.2.2.2D (Davis-Putnam SAT solver)
SAT11
    My implementation of Algorithm 7.2.2.2L (lookahead 3SAT solver)
SAT11K
    Change file to adapt SAT11 to clauses of arbitrary length
SAT12 and the companion program SAT12-ERP
    My implementation of a simple preprocessor for SAT
SAT13
    My implementation of Algorithm 7.2.2.2C (conflict-driven clause learning SAT
       solver)
```

(src)

They all are to be filtered via CWEB programs (CWEAVE and CTANGLE). Then you can compile both C and TeX files.

Here I did this for all these programs: `https://smt.st/current_tree/basics/SAT/Knuth`.

You can see that these solvers require Stanford Graphbase library, but this is just `gb_flip.h` module for PRNG. You can safely use srand()/rand().

So far, I rewrote only sat0w solver to down-to-earth C language: 23.2.

Also, D.Knuth's solver require slightly different input file format (not DIMACS), but the converters are available at the same Knuth's webpage.

2.4 Which SAT/SMT solver I should use?

Unfortunately, the whole field is still in its infancy. Solvers are continuously evolving and nothing stable yet.

A good idea is just try them all. Before using their APIs, first try to encode your problem in CNF or SMT-LIB format and just try all the solvers.

2.5 Yet another explanation of NP-problems

Various algorithms works ”slow” or ”fast” depending on the size of the input.

2.5.1 Constant time, O(1)

Time isn't dependent on size of input. This is like `string.size()` in C++ STL, given the fact that the implementation stores current string's size somewhere. Good: `.size()` method is fast, O(1), but during any modification of the string, a method(s) must update ”size” field.

2.5.2 Linear time, O(n)

Time is dependent on size of input, linearly. (Linear functions are functions which looks as lines on graph plot.) This is search algorithms on linked lists, strings. `strlen()` in C.

2.5.3 Hash tables and binary trees

These are used in associative arrays implementations. Using binary trees (`std::map`, `std::set` in C++ STL) is somewhat wasteful: you need too much memory, for each tree node. Also, search speed is a binary logarithm of the size of tree: $O(log(n))$ (because depth of binary tree is $log_2(size\_of\_tree)$). Good: keys are stored in sorted order, and you can retrieve them sorted while enumeration, by depth-first search of the binary tree.

Hash tables (`std::unordered_map` and `std::unordered_set` in C++ STL), on contrary, have constant access/insertion/deletion time ($O(1)$), and uses much less memory. This is because all key/value pairs are stored in a (big) table, and a hashed key is an index inside the table. Hash function should be good to make keys which are close to each other (100,101,102...) distributed uniformly across the table. However, you can't retrieve all keys from hash table in sorted form. It's a good idea to use hash tables instead of binary trees, if you don't need keys sorted.

2.5.4 EXPTIME

Time is dependent exponentially on size of input, $O(2^{p(n)})$ or just $O(2^n)$. This is bruteforce. When you try to break some cryptoalgorithm by bruteforcing and trying all possible inputs. For 8 bits of input, there are $2^8 = 256$ possible values, etc. Bruteforce can crack any cryptoalgorithm and solve almost any problem, but this is not very interesting, because it's extremely slow.

2.5.5 NP-problems

Finding correct inputs for CNF formula is an NP-problem, but you can also find them using simple bruteforce. In fact, SAT-solvers also do bruteforce, but the resulting search tree is extremely big. And to make search faster, SAT-solvers prune search tree as early as possible, but in unpredictable ways. This makes execution time of SAT-solvers also unpredictable. In fact, this is a problem, to predict how much time it will take to find a solution for CNF formula. No SAT/SMT solvers can say you ETA[45] of their work.

NP problems has no O-notation, meaning, there is no link between size (and contents) of input and execution time.

In contrast to other problems listed here (constant, linear, logarithmical, exponential), you can predict time of work by observing input values. You can predict, how long it will take to sort items by looking at a list of items to be sorted. This is a problem for NP-problems: you can't predict it.

Probably, you could devise faster algorithm to solve NP-problems, maybe working 1000 times faster, but still, it will work unpredictably.

Also, both SAT and SMT solvers are very capricious to input data. This is a rare case, when shotgun debugging [46] is justified!

2.5.6 P=NP?

One of the famous problem asking, if it's possible to solve NP-problems fast.

In other words, a scientist (or anyone else) who will find a way to solve NP-problems in predictable time (but faster than bruteforce/EXPTIME), will make a significant or revolutionary step in computer science.

In popular opinion, this will crack all cryptoalgorithms we currently use. In fact, this is not true. Some scientists, like Donald Knuth, believe, that there may be a way to solve NP-problems in polynomial time, but still too slow to be practical.

However, majority of scientists believe the humankind is not ready for such problems.

A list of many attempts to solve this problem: `https://www.win.tue.nl/~gwoegi/P-versus-NP.htm`.

2.5.7 What are SAT/SMT solvers?

To my taste, these are like APIs to NP-problems. Maybe not the best, but good enough. (Other APIs are, at least, CSPLIB, Prolog programming language...)

Because all NP-problems can be reduced (or converted) to SAT problems, or represented as SAT problems.

This is what I do in this book: I'm solving many well-known NP-problems using SAT/SMT solvers.

2.6 Origins of Tetris: a hypothesis

I don't know how Alexey Pajitnov invented Tetris, but here is my hypothesis.

There is a class of bin packing problems, see 10.2, 10.3, 22.7.

And pentomino figurines were quite popular as tests for packing problems, see the work of Solomon W. Golomb[47].

Another variation of bin packing problem is *online bin packing* – an algorithm is blind to the whole input, it only sees the current figurine and maybe the next one. But it have to pack all figurines to some box. *Online* in computer science means that an algorithm must work while observing only a small part of input. Almost like UNIX pipes...

> A good example is large-scale parcel packag-ing in modern logistics systems (Figure. 1), where parcels are mostly in regular cuboid shapes, and we would like to collectively pack them into rectangular bins of the standard dimension. Maximizing the storage use of bins effectively reduces the cost of inventorying, wrapping, transportation, and warehousing.

(`https://arxiv.org/pdf/2006.14978.pdf`)

And this is what Tetris players do all the time: they serve as a robot who take objects/figurines from a conveyor belt and pack them into boxes. A player sees only current object, and maybe a next one.

A player may think that figurines are dropping from the top, but take another look on it: figurines may be moving on conveyor belt from top to down. And not dropping at all, just moving slowly.

And how does the game reward you? When your packing is tight, with no voids (filled rows disappears).

[45] Estimated time of arrival

[46] `http://www.catb.org/jargon/html/S/shotgun-debugging.html` `https://en.wikipedia.org/wiki/Shotgun_debugging`

[47] `https://en.wikipedia.org/wiki/Polyominoes:_Puzzles,_Patterns,_Problems,_and_Packings`

This is what I observe almost daily in my real life near a house where I rent my flat: a backyard of an office of a highly popular Ukrainian delivery company "Nova Poshta"[48], a guy in a company's uniform packs all sorts of boxes into a van, when these boxes are coming from a hatch in a wall.

Pajitnov worked for *Dorodnitsyn Computing Centre of the Soviet Academy of Sciences* – not a videogame company. Probably he was busy with such an algorithm?

You may notice that virtually all puzzle games for smartphones that are popular now, are actually NP-problems. Someone can go even further: A human-based computation game or game with a purpose[49].

2.7 Halting problem

Judging by a source code of a function, can you say, will it terminate at some point, or will it spin into an infinite loop? Alan Turing famously proved that this is impossible (*Halting Problem*).

But let's see...

Let's say, you have a very basic 8-bit CPU, like 8080 or 6502. Say, there are only 5 8-bit registers, or 2 16-bit registers + one 8-bit register. Not uncommon for cheap CPUs of that era.

And there is no RAM attached. So all memory you can use is 40 bits. This is not uncommon for geeks of the time, when RAM chips were expensive and bought after CPU.

You can simulate such a basic CPU on any decent desktop computer. To track all states, you can use a dictionary. Key = values-from-all-registers-including-PC, value = boolean. The key has a size of 40 bits. The size of the whole dictionary is 2^{40} bits or 2^{32} bytes (just 4GB of RAM).

Now you can simulate any code running on this toy CPU using your desktop computer and track all states. When the code stops, you stop. When you hit a state that already encountered (checked using the dictionary), you stop and report an infinite loop.

Hence, to tell if the program will stop or not, you just need a computer with bigger RAM. To simulate a program running on your desktop computer with 4GB of RAM, you would need a computer with at least $2^{2^{64}}$ bits of RAM.

$log_{10}(2^{2^{64}}) \approx 5.55 \cdot 10^{18}$, but still theoretically possible.

These toy CPUs and computers are in fact LBAs (Linear bounded automaton) – a Turing machine with limited tape (or memory).

While Turing's Halting Problem is about Turing machine with infinite tape (or RAM).

[48] "New Post"

[49] https://en.wikipedia.org/wiki/Human-based_computation_game

Chapter 3

Equations

3.1 SMT-solver as a calculator

It's important to know that SMT-solver, like almost any PL, can be used as a generic calculator.

Listing 3.1: Using Z3Py API

```
#!/usr/bin/env python3

from z3 import *

s=Solver()
x,y,z=Ints('x y z')
s.add(x==2)
s.add(y==3)
s.add(z==x+y)
print (s.check())
print (s.model())
```

Listing 3.2: The result

```
sat
[z = 5, y = 3, x = 2]
```

Or in SMT-LIB 2.0 form:

```
(declare-fun x () (_ BitVec 16))
(declare-fun y () (_ BitVec 16))
(declare-fun z () (_ BitVec 16))

(assert (= x (_ bv2 16)))
(assert (= y (_ bv3 16)))

(assert (= z (bvadd x y)))

(check-sat)
(get-model)
```

Listing 3.3: The result

```
 % z3 -smt2 2.smt

sat
(model
  (define-fun z () (_ BitVec 16)
    #x0005)
  (define-fun y () (_ BitVec 16)
    #x0003)
  (define-fun x () (_ BitVec 16)
    #x0002)
)
```

Think about it as about electronic circuit – you can build an adder out from full-adders and logic gates. (We will do this soon: 23.3.1.) Then you turn on your circuit and get the result at the adder's output.

Now the *inverse calculator*. Can we solve the equation like $x + 3 = 17$?

```
(declare-fun x () (_ BitVec 16))

(assert (= (bvadd x (_ bv3 16)) (_ bv17 16)))

(check-sat)
(get-model)
```

Listing 3.4: The result

```
 % z3 -smt2 3.smt > 3.txt

sat
(model
  (define-fun x () (_ BitVec 16)
    #x000e)
)
```

(0xe is 14.)

3.2 Solving XKCD 287

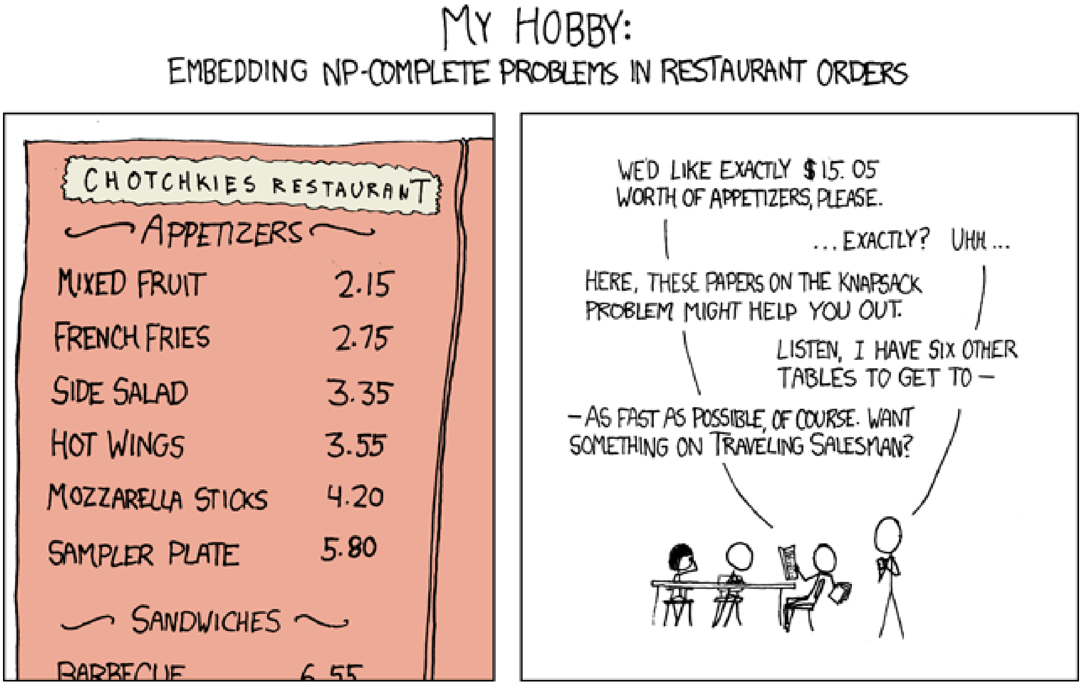

MY HOBBY:
EMBEDDING NP-COMPLETE PROBLEMS IN RESTAURANT ORDERS
CHOTCHKIES RESTAURANT
APPETIZERS
MIXED FRUIT 2.15
FRENCH FRIES 2.75
SIDE SALAD 3.35
HOT WINGS 3.55
MOZZARELLA STICKS 4.20
SAMPLER PLATE 5.80
SANDWICHES
BARBECUE 6.55
WE'D LIKE EXACTLY $15.05 WORTH OF APPETIZERS, PLEASE.
...EXACTLY? UHH...
HERE, THESE PAPERS ON THE KNAPSACK PROBLEM MIGHT HELP YOU OUT.
LISTEN, I HAVE SIX OTHER TABLES TO GET TO –
– AS FAST AS POSSIBLE, OF COURSE. WANT SOMETHING ON TRAVELING SALESMAN?

Figure 3.1: xkcd #287

(https://www.xkcd.com/287/)

The problem is to solve the following equation: $2.15a + 2.75b + 3.35c + 3.55d + 4.20e + 5.80f == 15.05$, where a..f are integers. So this is a linear diophantine equation.

```
#!/usr/bin/python3

from MK85 import *

```

```
s=MK85()
a=s.BitVec("a", 16)
b=s.BitVec("b", 16)
c=s.BitVec("c", 16)
d=s.BitVec("d", 16)
e=s.BitVec("e", 16)
f=s.BitVec("f", 16)

s.add(a<=10)
s.add(b<=10)
s.add(c<=10)
s.add(d<=10)
s.add(e<=10)
s.add(f<=10)

s.add(a*215 + b*275 + c*335 + d*355 + e*420 + f*580 == 1505)

while s.check():
    m=s.model()
    print (m)
    # block current solution and solve again:
    s.add(expr.Not(And(a==m["a"], b==m["b"], c==m["c"], d==m["d"], e==m["e"], f==m["f
        "])))
```

(The source code: https://smt.st/current_tree/equations/xkcd287/xkcd287_MK85.py)
There are just 2 solutions:

```
{'a': 7, 'c': 0, 'b': 0, 'e': 0, 'd': 0, 'f': 0}
{'a': 1, 'c': 0, 'b': 0, 'e': 0, 'd': 2, 'f': 1}
```

Wolfram Mathematica can solve the equation as well:

Listing 3.5: Wolfram Mathematica

```
In[]:= FindInstance[2.15 a + 2.75 b + 3.35 c + 3.55 d + 4.20 e + 5.80 f == 15.05 &&
        a >= 0 && b >= 0 && c >= 0 && d >= 0  && e >= 0 && f >= 0,
        {a, b, c, d, e, f}, Integers, 1000]

Out[]= {{a -> 1, b -> 0, c -> 0, d -> 2, e -> 0, f -> 1},
        {a -> 7, b -> 0, c -> 0, d -> 0, e -> 0, f -> 0}}
```

1000 means "find at most 1000 solutions", but only 2 are found. See also: http://reference.wolfram.com/language/ref/FindInstance.html.

Other ways to solve it: https://stackoverflow.com/questions/141779/solving-the-np-complete-problem-in-xkcd, http://www.explainxkcd.com/wiki/index.php/287:_NP-Complete.
The solution using Z3: https://smt.st/current_tree/equations/xkcd287/xkcd287_Z3.py)

3.2.1 Exporting the expression in SMT-LIB 2.x format

Add this:

```
--- xkcd287_Z3.py       2021-02-12 05:12:53.685491965 +0200
+++ xkcd287_Z3_sexpr.py 2021-07-14 02:17:55.920663814 +0300
@@ -6,6 +6,8 @@
 s = Solver()
 s.add(215*a + 275*b + 335*c + 355*d + 420*e + 580*f == 1505, a>=0, b>=0, c>=0, d>=0,
     e>=0, f>=0)

+print(s.sexpr())
+
 results=[]

 # enumerate all possible solutions:
```

You'll get:

```
(declare-fun f () Int)
(declare-fun e () Int)
(declare-fun d () Int)
(declare-fun c () Int)
(declare-fun b () Int)
(declare-fun a () Int)
(assert (= (+ (* 215 a) (* 275 b) (* 335 c) (* 355 d) (* 420 e) (* 580 f)) 1505))
(assert (>= a 0))
(assert (>= b 0))
(assert (>= c 0))
(assert (>= d 0))
(assert (>= e 0))
(assert (>= f 0))

[f = 0, b = 0, a = 7, c = 0, d = 0, e = 0]
[f = 1, b = 0, a = 1, c = 0, d = 2, e = 0]
results total= 2
```

That SMT-LIB 2.x code can be feed to another SMT solver, to check its speed, for example...

3.2.2 XKCD 287 in SMT-LIB 2.x format

Unlike integer type used before, this one uses bit vector:

```
; tested using MK85
; would work for Z3 if you uncomment "check-sat" and "get-model" and comment "get-all
    models"

(set-logic QF_BV)
(set-info :smt-lib-version 2.0)

(declare-fun a () (_ BitVec 16))
(declare-fun b () (_ BitVec 16))
(declare-fun c () (_ BitVec 16))
(declare-fun d () (_ BitVec 16))
(declare-fun e () (_ BitVec 16))
(declare-fun f () (_ BitVec 16))

(assert (bvult a #x0010))
(assert (bvult b #x0010))
(assert (bvult c #x0010))
(assert (bvult d #x0010))
(assert (bvult e #x0010))
(assert (bvult f #x0010))

(assert
        (=
                (bvadd
                        (bvmul (_ bv215 16) a)
                        (bvmul (_ bv275 16) b)
                        (bvmul (_ bv335 16) c)
                        (bvmul (_ bv355 16) d)
                        (bvmul (_ bv420 16) e)
                        (bvmul (_ bv580 16) f)
                )
                (_ bv1505 16)
        )
)

;(check-sat)
```

```
;(get-model)
(get-all-models)

; correct answer:

;(model
;         (define-fun a () (_ BitVec 16) (_ bv7 16)) ; 0x7
;         (define-fun b () (_ BitVec 16) (_ bv0 16)) ; 0x0
;         (define-fun c () (_ BitVec 16) (_ bv0 16)) ; 0x0
;         (define-fun d () (_ BitVec 16) (_ bv0 16)) ; 0x0
;         (define-fun e () (_ BitVec 16) (_ bv0 16)) ; 0x0
;         (define-fun f () (_ BitVec 16) (_ bv0 16)) ; 0x0
;)
;(model
;         (define-fun a () (_ BitVec 16) (_ bv1 16)) ; 0x1
;         (define-fun b () (_ BitVec 16) (_ bv0 16)) ; 0x0
;         (define-fun c () (_ BitVec 16) (_ bv0 16)) ; 0x0
;         (define-fun d () (_ BitVec 16) (_ bv2 16)) ; 0x2
;         (define-fun e () (_ BitVec 16) (_ bv0 16)) ; 0x0
;         (define-fun f () (_ BitVec 16) (_ bv1 16)) ; 0x1
;)
;Model count: 2
```

3.2.3 Other solutions

Some other solutions, including the one using Prolog: https://stackoverflow.com/q/141779.

3.2.4 A real life story

This happened in the middle of the night in Kiev, maybe in the second half of 2010s, in a local mall, that open around clock.

A pack of drunk youngsters heap up all sorts of groceries, beer cans, snacks before clerk and gave her a banknote of 100 Ukrainian hryvnias, asking to pick anything from that heap that they could buy with exactly this banknote, without change. They say that this banknote is all they have.

Surely, they couldn't realize how hard this problem is, as well as clerk, so they couldn't solve it without change...

3.3 Wood workshop, linear programming and Leonid Kantorovich

Let's say, you work for a wood workshop. You have a supply of rectangular wood workpieces, 6*13 inches, or meters, or whatever unit you use:

A workpiece (6*13 inches):

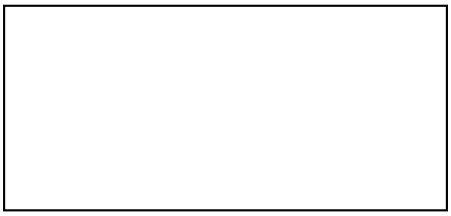
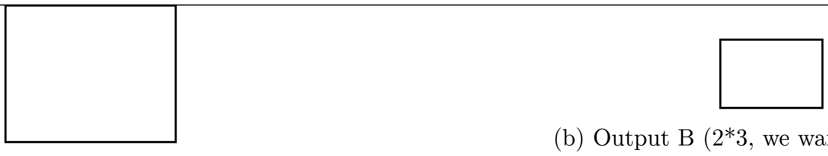


Figure 3.2: A 6*13 workpiece

You want to produce 800 rectangles of 4*5 size and 400 rectangles of 2*3 size:

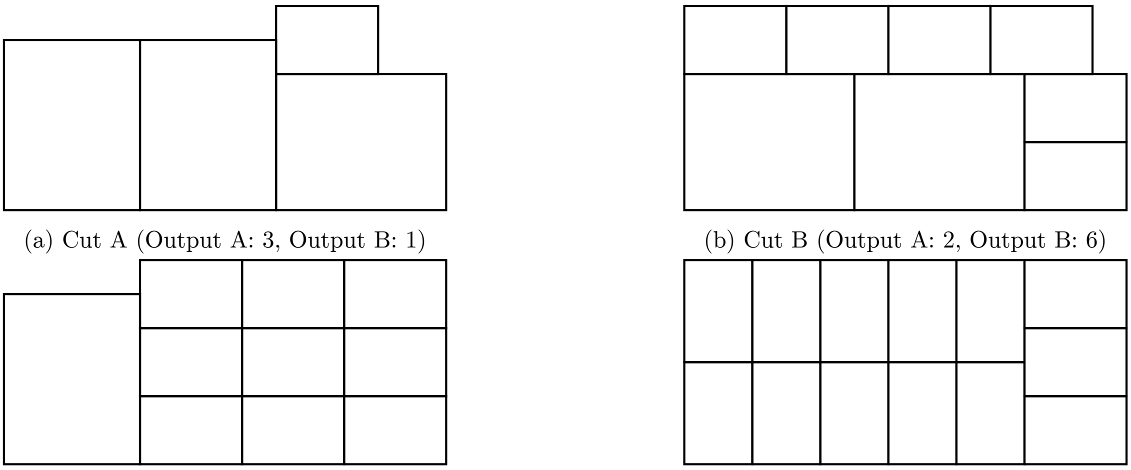


(a) Output A (4*5, we want 800 of these)

(b) Output B (2*3, we want 400 of these)

Figure 3.3: Rectangles we want

To cut a piece as A/B rectangles, you can cut a 6*13 workpiece in 4 ways. Or, to put it in another way, you can place A/B rectangles on 6*13 rectangle in 4 ways:



(a) Cut A (Output A: 3, Output B: 1)

(b) Cut B (Output A: 2, Output B: 6)

(c) Cut C (Output A: 1, Output B: 9)

(d) Cut D (Output A: 0, Output B: 13)

Figure 3.4: Four ways of cutting the workpiece

Now the problem. Which cuts are most efficient? You want to consume as little workpieces as possible.

This is optimization problem and I can solve it with Z3:

```
#!/usr/bin/env python3

from z3 import *

s=Optimize()

workpieces_total=Int('workpieces_total')
cut_A, cut_B, cut_C, cut_D=Ints('cut_A cut_B cut_C cut_D')
out_A, out_B=Ints('out_A out_B')

s.add(workpieces_total==cut_A+cut_B+cut_C+cut_D)

s.add(cut_A>=0)
s.add(cut_B>=0)
s.add(cut_C>=0)
s.add(cut_D>=0)

s.add(out_A==cut_A*3 + cut_B*2 + cut_C*1)
s.add(out_B==cut_A*1 + cut_B*6 + cut_C*9 + cut_D*13)

s.add(out_A==800)
s.add(out_B==400)

s.minimize(workpieces_total)

print (s.check())
print (s.model())
```

```
sat
[cut_B = 25,
 cut_D = 0,
 cut_A = 250,
 out_B = 400,
 out_A = 800,
 workpieces_total = 275,
 cut_C = 0]
```

So you want to cut 250 workpieces in A's way and 25 pieces in B's way, this is the most optimal way.

Also, the problem is small enough to be solved by my toy bit-blaster MK85, (thanks to the Open-WBO MaxSAT solver):

```
(declare-fun workpieces_total () (_ BitVec 16))
(declare-fun cut_A () (_ BitVec 16))
(declare-fun cut_B () (_ BitVec 16))
(declare-fun cut_C () (_ BitVec 16))
(declare-fun cut_D () (_ BitVec 16))
(declare-fun out_A () (_ BitVec 16))
(declare-fun out_B () (_ BitVec 16))

(assert (bvuge cut_A (_ bv0 16)))
(assert (bvuge cut_B (_ bv0 16)))
(assert (bvuge cut_C (_ bv0 16)))
(assert (bvuge cut_D (_ bv0 16)))

(assert (bvuge out_A (_ bv800 16)))
(assert (bvuge out_B (_ bv400 16)))

(assert (= workpieces_total (bvadd cut_A cut_B cut_C cut_D)))

(assert (= out_A (bvadd
                (bvmul_no_overflow cut_A (_ bv3 16))
                (bvmul_no_overflow cut_B (_ bv2 16))
                cut_C
                )
        )
)

(assert (= out_B (bvadd
                cut_A
                (bvmul_no_overflow cut_B (_ bv6 16))
                (bvmul_no_overflow cut_C (_ bv9 16))
                (bvmul_no_overflow cut_D (_ bv13 16))
                )
        )
)

(minimize workpieces_total)

(check-sat)
(get-model)
```

Listing 3.6: The result

```
sat
(model
        (define-fun cut_A () (_ BitVec 16) (_ bv250 16)) ; 0xfa
        (define-fun cut_B () (_ BitVec 16) (_ bv25 16)) ; 0x19
        (define-fun cut_C () (_ BitVec 16) (_ bv0 16)) ; 0x0
        (define-fun cut_D () (_ BitVec 16) (_ bv0 16)) ; 0x0
        (define-fun out_A () (_ BitVec 16) (_ bv800 16)) ; 0x320
        (define-fun out_B () (_ BitVec 16) (_ bv400 16)) ; 0x190
        (define-fun workpieces_total () (_ BitVec 16) (_ bv275 16)) ; 0x113
)
```

The task I solved I've found in Leonid Kantorovich's book "The Best Uses of Economic Resources" (1959). And these are 5 pages with the task and solution [1] (in Russian).

Leonid Kantorovich was indeed consulting plywood factory in 1939 about optimal use of materials. And this is how linear programming (LP) and ILP[2] has emerged.

See also: https://en.wikipedia.org/wiki/Guillotine_cutting#Finding_an_optimal_cutting-pattern.

3.4 Puzzle with animals

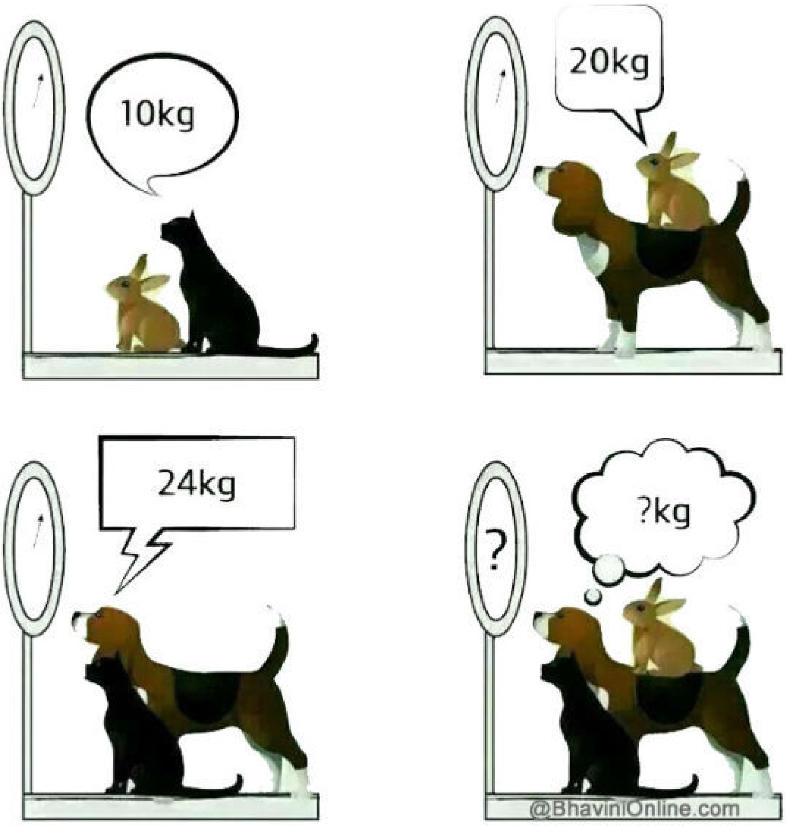


Figure 3.5: Found this elsewhere

This puzzle[3] can be solved easily, even with my toy MK85 solver:

```
; https://www.bhavinionline.com/2014/10/whatsapp-picture-puzzle-find-weight-rabbit-
   cat-dog/
(set-logic QF_BV)
(set-info :smt-lib-version 2.0)

(declare-fun dog () (_ BitVec 16))
(declare-fun cat () (_ BitVec 16))
(declare-fun rabbit () (_ BitVec 16))

(assert (= (bvadd rabbit cat) (_ bv10 16)))
(assert (= (bvadd dog rabbit) (_ bv20 16)))
(assert (= (bvadd dog cat) (_ bv24 16)))
```

[1] https://smt.st/current_tree/equations/kantorovich/from_book

[2] Integer Linear Programming

[3] https://www.bhavinionline.com/2014/10/whatsapp-picture-puzzle-find-weight-rabbit-cat-dog/

```
(check-sat)
(get-model)
;(get-all-models)
```

```
sat
(model
        (define-fun cat () (_ BitVec 16) (_ bv7 16)) ; 0x7
        (define-fun dog () (_ BitVec 16) (_ bv17 16)) ; 0x11
        (define-fun rabbit () (_ BitVec 16) (_ bv3 16)) ; 0x3
)
```

However, one can argue that problem like that can be easily solved with bruteforce. I'll make it more realistic and harder.

Let's say, there are 4 objects, each has a mass measured in some *units*, in milligrams or even micrograms. Each object's mass is 64-bit value.

We can use a weighting scale to measure object's mass, but as it happens in the real world, the scale is not precise. We use two type of scales: the first can measure object down to $10^3 \cdot unit$, the second down to $10^4 \cdot unit$.

We do our measures and we set mass in some range: [xxxx...000 - xxxx...999] and [xxxx...0000 - xxxx...9990].

```
(set-logic QF_BV)
(set-info :smt-lib-version 2.0)

(declare-fun a () (_ BitVec 64))
(declare-fun b () (_ BitVec 64))
(declare-fun c () (_ BitVec 64))
(declare-fun d () (_ BitVec 64))

(assert (bvugt (bvadd a b c) (_ bv3759470599491624000 64)))
(assert (bvult (bvadd a b c) (_ bv3759470599491624999 64)))

(assert (bvugt (bvadd c d) (_ bv2131460860763223000 64)))
(assert (bvult (bvadd c d) (_ bv2131460860763223999 64)))

(assert (bvugt (bvadd a b d) (_ bv5028681613540470000 64)))
(assert (bvult (bvadd a b d) (_ bv5028681613540479999 64)))

(assert (bvugt (bvadd b c) (_ bv2079657009873280000 64)))
(assert (bvult (bvadd b c) (_ bv2079657009873289999 64)))

(assert (bvugt (bvadd a c) (_ bv2110938512975520000 64)))
(assert (bvult (bvadd a c) (_ bv2110938512975529999 64)))

(check-sat)
(get-model)
```

Even my toy MK85 can solve this:

```
sat
(model
        (define-fun a () (_ BitVec 64) (_ bv1679813589618343937 64)) ; 0
           x174fe58f4a134001
        (define-fun b () (_ BitVec 64) (_ bv1648532086516097041 64)) ; 0
           x16e0c332c9a02011
        (define-fun c () (_ BitVec 64) (_ bv431124923357184020 64)) ; 0
           x5fba9d56685c014
        (define-fun d () (_ BitVec 64) (_ bv1700335937406039020 64)) ; 0
           x1798ce86bbeaf3ec
)
```

Almost correct, considering the fact that scales are not precise to the last *unit*.

This is the data I used for the example:

Listing 3.7: Wolfram Mathematica code

```
In[]:= a=RandomInteger[(2^62)-1]
Out[]= 1679813589618336900

In[]:= b=RandomInteger[(2^62)-1]
Out[]= 1648532086516099370

In[]:= c=RandomInteger[(2^62)-1]
Out[]= 431124923357188627

In[]:= d=RandomInteger[(2^62)-1]
Out[]= 1700335937406035054

In[]:= a+b+c
Out[]= 3759470599491624897

In[]:= a+b+d
Out[]= 5028681613540471324

In[]:= c+d
Out[]= 2131460860763223681

In[]:= b+c
Out[]= 2079657009873287997

In[]:= BaseForm[{a,b,c,d},16]
Out[]//BaseForm= {Subscript[174fe58f4a132484, 16],Subscript[16e0c332c9a0292a, 16],
    Subscript[5fba9d56685d213, 16],Subscript[1798ce86bbeae46e, 16]}
```

3.5 Subset sum

> In computer science, the subset sum problem is an important problem in complexity theory and cryptography. The problem is this: given a set (or multiset) of integers, is there a non-empty subset whose sum is zero? For example, given the set $\{-7, -3, -2, 5, 8\}$, the answer is yes because the subset $\{-3, -2, 5\}$ sums to zero.

(https://en.wikipedia.org/wiki/Subset_sum_problem)

It's expressible easily as 0-1 ILP problem:

```
from z3 import *

set=[-7, -3, -2, 5, 8]

set_len=len(set)

vars=[Int('vars_%d' % i) for i in range(set_len)]

s=Solver()
rt=[]

for i in range(set_len):
    rt.append(vars[i]*set[i])
    s.add(Or(vars[i]==0, vars[i]==1)) # like bools

# rt here is [vars_0*-7, vars_1*-3, vars_2*-2, vars_3*5, vars_4*8]

s.add(sum(rt)==0)
```

```
s.add(sum(vars)>=1) # subset must not be empty

if s.check()==False:
    print "unsat"
    exit(0)

m=s.model()

for i in range(set_len):
    if m[vars[i]].as_long()==1:
        print set[i],
print ""
```

Listing 3.8: The result

```
-3 -2 5
```

3.6 Art of problem solving

http://artofproblemsolving.com/wiki/index.php?title=2017_AMC_12A_Problems/Problem_2:

```
The sum of two nonzero real numbers is 4 times their product.
What is the sum of the reciprocals of the two numbers?
```

We're going to solve this over real numbers:

```
from z3 import *

x, y = Reals('x y')

s=Solver()

s.add(x>0)
s.add(y>0)

s.add(x+y == 4*x*y)

print s.check()
m=s.model()
print "the model:"
print m
print "the answer:", m.evaluate (1/x + 1/y)
```

Instead of pulling values from the model and then compute the final result on Python's side, we can evaluate an expression $(\frac{1}{x} + \frac{1}{y})$ *inside* the model we've got:

```
sat
the model:
[x = 1, y = 1/3]
the answer: 4
```

3.7 Yet another explanation of modulo inverse using SMT-solvers

Untitled collection of math notes[4]has a part about modulo arithmetics and modulo inverse.

By which constant we must multiply a random number, so that the result would be as if we divided them by 3?

```
from z3 import *

m=BitVec('m', 32)
```

[4] https://xn--mxacd.xn--qxam/math-notes.pdf

```
s=Solver()

# wouldn't work for 10, etc
divisor=3

# random constant, must be divisible by divisor:
const=(0x1234567*divisor)

s.add(const*m == const/divisor)

print s.check()
print "%x" % s.model()[m].as_long()
```

The magic number is:

```
sat
aaaaaaab
```

Indeed, this is modulo inverse of 3 modulo 2^{32}: https://www.wolframalpha.com/input/?i=PowerMod%5B3,-1,2%5E32%5D.

Let's check using my calculator:

```
[3] 123456*0xaaaaaaab
[3] (unsigned) 353492988371136 0x141800000a0c0 0
    b1010000011000000000000000000000001010000011000000
[4] 123456/3
[4] (unsigned) 41152 0xa0c0 0b1010000011000000
```

The problem is simple enough to be solved using MK85:

```
; find modulo inverse
; checked with Z3 and MK85
(set-logic QF_BV)
(set-info :smt-lib-version 2.0)

(declare-fun m () (_ BitVec 16))
(declare-fun a () (_ BitVec 16))
(declare-fun b () (_ BitVec 16))

(assert (= a (bvudiv #x1236 #x0003)))
(assert (= b (bvmul #x1236 m)))

(assert (= a b))

; without this constraint, two results would be generated (with MSB=1 and MSB=0),
; but we need only one indeed, MSB of m has no effect of multiplication here
; and SMT-solver offers two solutions
(assert (= (bvand m #x8000) #x0000))

(check-sat)
(get-model)
;(get-all-models)
```

```
sat
(model
        (define-fun m () (_ BitVec 16) (_ bv10923 16)) ; 0x2aab
        (define-fun a () (_ BitVec 16) (_ bv1554 16)) ; 0x612
        (define-fun b () (_ BitVec 16) (_ bv1554 16)) ; 0x612
)
```

However, it wouldn't work for 10, because there are no modulo inverse of 10 modulo 2^{32}, SMT solver would give "unsat".

3.8 School-level equation: KLEE

Let's revisit school-level system of equations from (2.2.2).

We will force KLEE to find a path, where all the constraints are satisfied:

```
#include <assert.h>
#include "klee.h"

int main()
{
        int circle, square, triangle;

        klee_make_symbolic(&circle, sizeof circle, "circle");
        klee_make_symbolic(&square, sizeof square, "square");
        klee_make_symbolic(&triangle, sizeof triangle, "triangle");

        if (circle+circle!=10) return 0;
        if (circle*square+square!=12) return 0;
        if (circle*square-triangle*circle!=circle) return 0;

        // all constraints should be satisfied at this point
        // force KLEE to produce .err file:
        assert(0);
};
```

```
% clang -emit-llvm -c -g -O0 -Xclang -disable-O0-optnone -I /tmp/klee_src/include/
   klee/ klee_eq1.c

...

% time klee --libc=uclibc --optimize --use-query-log=solver:smt2 klee_eq1.bc
KLEE: done: total instructions = 11794
KLEE: done: completed paths = 2
KLEE: done: partially completed paths = 1
KLEE: done: generated tests = 3
```

Let's find out, where `klee_assert()` has been triggered:

```
% ls klee-last/*err
klee-last/test000001.external.err

% ktest-tool klee-last/test000001.ktest | grep uint
object 0: uint: 5
object 1: uint: 2
object 2: uint: 1
```

This is indeed correct solution to the system of equations.

But 3 tests generated. What are others? Others are *partial* solutions, which leads to one of *return* statements. See explanation: 8.2.2.

KLEE has *intrinsic* `klee_assume()` which tells KLEE to cut path if some constraint is not satisfied. So we can rewrite our example in such cleaner way:

```
#include <assert.h>
#include "klee.h"

int main()
{
        int circle, square, triangle;

        klee_make_symbolic(&circle, sizeof circle, "circle");
        klee_make_symbolic(&square, sizeof square, "square");
        klee_make_symbolic(&triangle, sizeof triangle, "triangle");

        klee_assume (circle+circle==10);
        klee_assume (circle*square+square==12);
        klee_assume (circle*square-triangle*circle==circle);

        // all constraints should be satisfied at this point
        // force KLEE to produce .err file:
        assert(0);
};
```

Interestingly, we can dump SMT-LIB 2 file here and see its contents:

```
% time klee --libc=uclibc --optimize --use-query-log=solver:smt2 klee_eq2.bc

KLEE: done: total instructions = 11652
KLEE: done: completed paths = 0
KLEE: done: partially completed paths = 1
KLEE: done: generated tests = 1
```

```
; Query 0 -- Type: InitialValues, Instructions: 11628
(set-option :produce-models true)
(set-logic QF_AUFBV )
(declare-fun circle () (Array (_ BitVec 32) (_ BitVec 8) ) )
(assert (=  (_ bv5 32) (concat  (select  circle (_ bv3 32) ) (concat  (select  circle
    (_ bv2 32) ) (concat  (select  circle (_ bv1 32) ) (select  circle (_ bv0 32) ) )
    ) ) ) )
(check-sat)
(get-value ( (select circle (_ bv0 32) ) ) )
(get-value ( (select circle (_ bv1 32) ) ) )
(get-value ( (select circle (_ bv2 32) ) ) )
(get-value ( (select circle (_ bv3 32) ) ) )
(exit)
;   OK -- Elapsed: 2.367557e-03s
;   Solvable: true
;     circle = [5,0,0,0]

; Query 1 -- Type: InitialValues, Instructions: 11638
(set-option :produce-models true)
(set-logic QF_AUFBV )
(declare-fun square () (Array (_ BitVec 32) (_ BitVec 8) ) )
(assert (let ( (?B1 (concat  (select  square (_ bv3 32) ) (concat  (select  square (_
    bv2 32) ) (concat  (select  square (_ bv1 32) ) (select  square (_ bv0 32) ) ) )
   ) ) ) (=  (_ bv12 32) (bvadd  (bvmul  (_ bv5 32) ?B1 ) ?B1 ) ) ) )
(check-sat)
(get-value ( (select square (_ bv0 32) ) ) )
(get-value ( (select square (_ bv1 32) ) ) )
(get-value ( (select square (_ bv2 32) ) ) )
(get-value ( (select square (_ bv3 32) ) ) )
(exit)
;   OK -- Elapsed: 3.219298e-02s
;   Solvable: true
;     square = [2,0,0,0]

; Query 2 -- Type: InitialValues, Instructions: 11651
(set-option :produce-models true)
(set-logic QF_AUFBV )
(declare-fun square () (Array (_ BitVec 32) (_ BitVec 8) ) )
(declare-fun triangle () (Array (_ BitVec 32) (_ BitVec 8) ) )
(assert (let ( (?B1 (concat  (select  square (_ bv3 32) ) (concat  (select  square (_
    bv2 32) ) (concat  (select  square (_ bv1 32) ) (select  square (_ bv0 32) ) ) )
   ) ) ) (and  (=  (_ bv5 32) (bvmul  (_ bv5 32) (bvsub  ?B1 (concat  (select
   triangle (_ bv3 32) ) (concat  (select  triangle (_ bv2 32) ) (concat  (select
   triangle (_ bv1 32) ) (select  triangle (_ bv0 32) ) ) ) ) ) ) ) (=  (_ bv12 32) (
   bvadd  (bvmul  (_ bv5 32) ?B1 ) ?B1 ) ) ) ) )
(check-sat)
(get-value ( (select square (_ bv0 32) ) ) )
(get-value ( (select square (_ bv1 32) ) ) )
(get-value ( (select square (_ bv2 32) ) ) )
(get-value ( (select square (_ bv3 32) ) ) )
(get-value ( (select triangle (_ bv0 32) ) ) )
(get-value ( (select triangle (_ bv1 32) ) ) )
(get-value ( (select triangle (_ bv2 32) ) ) )
(get-value ( (select triangle (_ bv3 32) ) ) )
(exit)
;   OK -- Elapsed: 1.125595e-02s
;   Solvable: true
;     square = [2,0,0,0]
;     triangle = [1,0,0,0]
```

There are 3 queries, each for each path. I can try them all separately.

Listing 3.9: First query

```
 % z3 1.smt2
sat
(((select circle (_ bv0 32) ) #x05))
(((select circle (_ bv1 32) ) #x00))
(((select circle (_ bv2 32) ) #x00))
(((select circle (_ bv3 32) ) #x00))
```

Listing 3.10: Second query

```
 % z3 2.smt2
sat
(((select square (_ bv0 32) ) #x02))
(((select square (_ bv1 32) ) #x00))
(((select square (_ bv2 32) ) #x00))
(((select square (_ bv3 32) ) #x00))
```

Listing 3.11: Third query

```
% z3 3.smt2
sat
(((select square (_ bv0 32) ) #x02))
(((select square (_ bv1 32) ) #x00))
(((select square (_ bv2 32) ) #x00))
(((select square (_ bv3 32) ) #x00))
(((select triangle (_ bv0 32) ) #x01))
(((select triangle (_ bv1 32) ) #x00))
(((select triangle (_ bv2 32) ) #x00))
(((select triangle (_ bv3 32) ) #x00))
```

Aha, here the solution in surfacing gradually. First, KLEE determines the correct value for ’circle’, then for ’square’, then for ’triangle’. Each 32-bit variable of *integer* type is represented by 4-bytes tuple. Maybe there is some reason for this.

3.9 School-level equation: CBMC

Doing this again with CBMC:

```
#include <assert.h>

int main()
```

```
{
        int circle, square, triangle;

        if (circle+circle!=10)
                return 0;
        if (circle*square+square!=12)
                return 0;
        if (circle*square-triangle*circle!=circle)
                return 0;

        // all constraints should be satisfied at this point
        assert(0);
        return 1;
};
```

```
% cbmc --trace --function main eq1.c

...
** Results:
eq1.c function main
[main.assertion.1] line 15 assertion 0: FAILURE

Trace for main.assertion.1:

State 21 file eq1.c function main line 5 thread 0
----------------------------------------------------
  circle=5 (00000000 00000000 00000000 00000101)

State 22 file eq1.c function main line 5 thread 0
----------------------------------------------------
  square=-2147483646 (10000000 00000000 00000000 00000010)

State 23 file eq1.c function main line 5 thread 0
----------------------------------------------------
  triangle=-2147483647 (10000000 00000000 00000000 00000001)
...
```

Ouch! This is correct solution, but for signed integers.

I can fix it:

```
#include <assert.h>

int main()
{
        int circle, square, triangle;

        if (circle+circle!=10)
                return 0;
        if (circle*square+square!=12)
                return 0;
        if (circle*square-triangle*circle!=circle)
                return 0;
        if (circle<0)
                return 0;
        if (square<0)
                return 0;
        if (triangle<0)
                return 0;

        // all constraints should be satisfied at this point
        assert(0);
        return 1;
};
```

```
 % cbmc --trace --function main eq2.c

...

** Results:
eq2.c function main
[main.assertion.1] line 21 assertion 0: FAILURE

Trace for main.assertion.1:

State 21 file eq2.c function main line 5 thread 0
----------------------------------------------------
  circle=5 (00000000 00000000 00000000 00000101)

State 22 file eq2.c function main line 5 thread 0
----------------------------------------------------
  square=2 (00000000 00000000 00000000 00000010)

State 23 file eq2.c function main line 5 thread 0
----------------------------------------------------
  triangle=1 (00000000 00000000 00000000 00000001)

...
```

Or let's just declare unsigned integers, this will also work:

```
#include <assert.h>

int main()
{
        unsigned int circle, square, triangle;

        if (circle+circle!=10)
                return 0;
        if (circle*square+square!=12)
                return 0;
        if (circle*square-triangle*circle!=circle)
                return 0;

        // all constraints should be satisfied at this point
        assert(0);
};
```

CBMC can also export SMT-LIB 2 file:

```
% cbmc --trace --function main --smt2 --outfile 1.smt eq3.c
```

The core of it is:

```
; find_symbols
(declare-fun |main::1::circle!0@1#1| () (_ BitVec 32))
; set_to true
(assert (= |goto_symex::&92;guard#1| (not (= (bvadd |main::1::circle!0@1#1| |main
    ::1::circle!0@1#1|) (_ bv10 32)))))

; set_to true (equal)
(define-fun |main#return_value!0#1| () (_ BitVec 32) (_ bv0 32))

; find_symbols
(declare-fun |main::1::square!0@1#1| () (_ BitVec 32))
; set_to true
```

```
(assert (= |goto_symex::&92;guard#2| (not (= (bvadd (bvmul |main::1::circle!0@1#1| |
    main::1::square!0@1#1|) |main::1::square!0@1#1|) (_ bv12 32)))))

; set_to true (equal)
(define-fun |main#return_value!0#2| () (_ BitVec 32) (_ bv0 32))

; find_symbols
(declare-fun |main::1::triangle!0@1#1| () (_ BitVec 32))
; set_to true
(assert (= |goto_symex::&92;guard#3| (not (= (bvadd (bvmul |main::1::circle!0@1#1| |
    main::1::square!0@1#1|) (bvneg (bvmul |main::1::circle!0@1#1| |main::1::triangle!0
    @1#1|))) |main::1::circle!0@1#1|))))

; set_to true (equal)
(define-fun |main#return_value!0#3| () (_ BitVec 32) (_ bv0 32))

; convert
(define-fun |B9| () Bool (= |main::1::circle!0@1#1| |main::1::circle!0@1#1|))

; convert
(define-fun |B10| () Bool (= |main::1::square!0@1#1| |main::1::square!0@1#1|))

; convert
(define-fun |B11| () Bool (= |main::1::triangle!0@1#1| |main::1::triangle!0@1#1|))

; convert
(define-fun |B12| () Bool (= (bvadd |main::1::circle!0@1#1| |main::1::circle!0@1#1|)
    (_ bv10 32)))

; convert
(define-fun |B13| () Bool (= (bvadd (bvmul |main::1::circle!0@1#1| |main::1::square!0
    @1#1|) |main::1::square!0@1#1|) (_ bv12 32)))

; convert
(define-fun |B14| () Bool (= (bvadd (bvmul |main::1::circle!0@1#1| |main::1::square!0
    @1#1|) (bvneg (bvmul |main::1::circle!0@1#1| |main::1::triangle!0@1#1|))) |main
    ::1::circle!0@1#1|))
```

(Full version: https://smt.st/current_tree/equations/CBMC_eq/1.smt.)

I can run it separately in SMT solver:

```
...
((|main::1::circle!0@1#1| #x00000005))
((|main::1::square!0@1#1| #x00000002))
((|main::1::triangle!0@1#1| #x00000001))
```

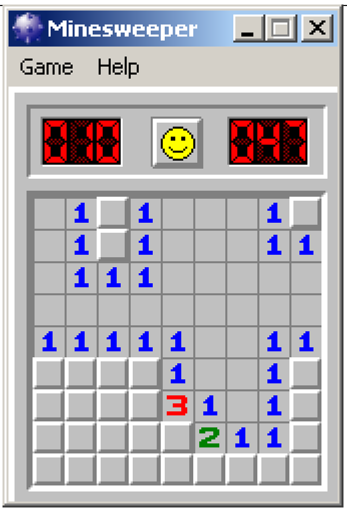
This is the correct solution indeed.

3.10 Minesweeper

3.10.1 Cracking Minesweeper with SMT solver

For those who are not very good at playing Minesweeper (like me), it's possible to predict bombs' placement without touching debugger.

Here is a clicked somewhere and I see revealed empty cells and cells with known number of "neighbours":



What we have here, actually? Hidden cells, empty cells (where bombs are not present), and empty cells with numbers, which shows how many bombs are placed nearby.

The method

Unlike many other examples, where our goal is to find a solution, here we use the fact that an instance is unsolvable (unsat).

Here is what we can do: we will try to place a bomb to all possible hidden cells and ask Z3 SMT solver, if it can disprove the very fact that the bomb can be placed there.

Take a look at this fragment. "?" mark is for hidden cell, "." is for empty cell, number is a number of neighbours.

	C1	C2	C3
R1	?	?	?
R2	?	3	.
R3	?	1	.

So there are 5 hidden cells. We will check each hidden cell by placing a bomb there. Let's first pick top/left cell:

	C1	C2	C3
R1	*	?	?
R2	?	3	.
R3	?	1	.

Then we will try to solve the following system of equations (*RrCc* is cell of row *r* and column *c*):

- R1C1=1 (because we placed bomb at R1C1)
- R2C1+R2C2+R2C3+R3C1+R3C2+R3C3=1 (because we have "1" at R3C2)
- R1C1+R1C2+R1C3+R2C1+R2C2+R2C3+R3C1+R3C2+R3C3=3 (because we have "3" at R2C2)
- R1C2+R1C3+R2C2+R2C3+R3C2+R3C3=0 (because we have "." at R2C3)
- R2C2+R2C3+R3C2+R3C3=0 (because we have "." at R3C3)

As it turns out, this system of equations is satisfiable, so there could be a bomb at this cell. But this information is not interesting to us, since we want to find cells we can freely click on. And we will try another one. And if the equation will be unsatisfiable, that would imply that a bomb cannot be there and we can click on it.

(This is so called "0-1 integer linear programming", since unknown variables are limited to 0/1.)

The code

```
#!/usr/bin/python3

known=[
"01?10001?",
"01?100011",
"011100000",
"000000000",
"111110011",
"????1001?",
"????3101?",
"?????211?",
"?????????"]

from z3 import *
import sys

WIDTH=len(known[0])
HEIGHT=len(known)

print ("WIDTH=", WIDTH, "HEIGHT=", HEIGHT)

def chk_bomb(row, col):

    s=Solver()

    cells=[[Int('r%d_c%d' % (r,c)) for c in range(WIDTH+2)] for r in range(HEIGHT+2)]

    # make border
    for c in range(WIDTH+2):
        s.add(cells[0][c]==0)
        s.add(cells[HEIGHT+1][c]==0)
    for r in range(HEIGHT+2):
        s.add(cells[r][0]==0)
        s.add(cells[r][WIDTH+1]==0)

    for r in range(1,HEIGHT+1):
        for c in range(1,WIDTH+1):
            # otherwise -1 is possible, etc:
            s.add(Or(cells[r][c]==0, cells[r][c]==1))

            t=known[r-1][c-1]
            if t in "012345678":
                s.add(cells[r][c]==0)
                # we need empty border so the following expression would be able to
                   work for all possible cases:
                expr=cells[r-1][c-1] + cells[r-1][c] + cells[r-1][c+1] + cells[r][c
                   -1] + cells[r][c+1] + cells[r+1][c-1] + cells[r+1][c] + cells[r
                   +1][c+1]==int(t)
                if False:
                    print (expr)
                s.add(expr)

    # place bomb:
    s.add(cells[row][col]==1)

    if s.check()==unsat:
        print ("row=%d col=%d, unsat!" % (row, col))

# enumerate all hidden cells:
for r in range(1,HEIGHT+1):
    for c in range(1,WIDTH+1):
        if known[r-1][c-1]=="?":
            chk_bomb(r, c)
```

The code is almost self-explanatory. We need a border for the same reason, why Conway's "Game of Life" implementations also has a border (to make calculation function simpler). Whenever we know that the cell is free of bomb, we put zero there. Whenever we know number of neighbours, we add a constraint, again, just like in ”Game of Life”: number of neighbours must be equal to the number we have seen in the Minesweeper. Then we place bomb somewhere and check.

Based on map in the example, my script first tries to put a bomb at row=1 col=3 (first ”?”). And this system of equations being generated:

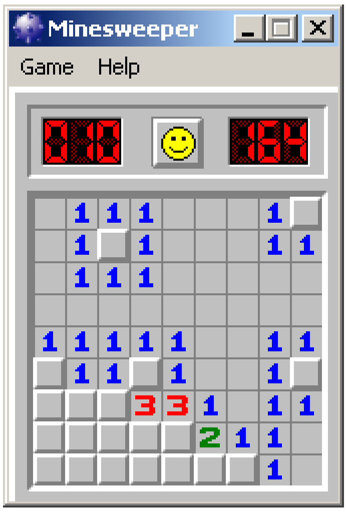
```
r0_c0 + r0_c1 + r0_c2 + r1_c0 + r1_c2 + r2_c0 + r2_c1 + r2_c2 == 0
r0_c1 + r0_c2 + r0_c3 + r1_c1 + r1_c3 + r2_c1 + r2_c2 + r2_c3 == 1
r0_c3 + r0_c4 + r0_c5 + r1_c3 + r1_c5 + r2_c3 + r2_c4 + r2_c5 == 1
r0_c4 + r0_c5 + r0_c6 + r1_c4 + r1_c6 + r2_c4 + r2_c5 + r2_c6 == 0
r0_c5 + r0_c6 + r0_c7 + r1_c5 + r1_c7 + r2_c5 + r2_c6 + r2_c7 == 0
r0_c6 + r0_c7 + r0_c8 + r1_c6 + r1_c8 + r2_c6 + r2_c7 + r2_c8 == 0
r0_c7 + r0_c8 + r0_c9 + r1_c7 + r1_c9 + r2_c7 + r2_c8 + r2_c9 == 1
r1_c0 + r1_c1 + r1_c2 + r2_c0 + r2_c2 + r3_c0 + r3_c1 + r3_c2 == 0
r1_c1 + r1_c2 + r1_c3 + r2_c1 + r2_c3 + r3_c1 + r3_c2 + r3_c3 == 1
r1_c3 + r1_c4 + r1_c5 + r2_c3 + r2_c5 + r3_c3 + r3_c4 + r3_c5 == 1
r1_c4 + r1_c5 + r1_c6 + r2_c4 + r2_c6 + r3_c4 + r3_c5 + r3_c6 == 0
r1_c5 + r1_c6 + r1_c7 + r2_c5 + r2_c7 + r3_c5 + r3_c6 + r3_c7 == 0
r1_c6 + r1_c7 + r1_c8 + r2_c6 + r2_c8 + r3_c6 + r3_c7 + r3_c8 == 0
r1_c7 + r1_c8 + r1_c9 + r2_c7 + r2_c9 + r3_c7 + r3_c8 + r3_c9 == 1
r1_c8 + r1_c9 + r1_c10 + r2_c8 + r2_c10 + r3_c8 + r3_c9 + r3_c10 == 1
r2_c0 + r2_c1 + r2_c2 + r3_c0 + r3_c2 + r4_c0 + r4_c1 + r4_c2 == 0
r2_c1 + r2_c2 + r2_c3 + r3_c1 + r3_c3 + r4_c1 + r4_c2 + r4_c3 == 1
r2_c2 + r2_c3 + r2_c4 + r3_c2 + r3_c4 + r4_c2 + r4_c3 + r4_c4 == 1
r2_c3 + r2_c4 + r2_c5 + r3_c3 + r3_c5 + r4_c3 + r4_c4 + r4_c5 == 1
r2_c4 + r2_c5 + r2_c6 + r3_c4 + r3_c6 + r4_c4 + r4_c5 + r4_c6 == 0
r2_c5 + r2_c6 + r2_c7 + r3_c5 + r3_c7 + r4_c5 + r4_c6 + r4_c7 == 0
r2_c6 + r2_c7 + r2_c8 + r3_c6 + r3_c8 + r4_c6 + r4_c7 + r4_c8 == 0
r2_c7 + r2_c8 + r2_c9 + r3_c7 + r3_c9 + r4_c7 + r4_c8 + r4_c9 == 0
r2_c8 + r2_c9 + r2_c10 + r3_c8 + r3_c10 + r4_c8 + r4_c9 + r4_c10 == 0
r3_c0 + r3_c1 + r3_c2 + r4_c0 + r4_c2 + r5_c0 + r5_c1 + r5_c2 == 0
r3_c1 + r3_c2 + r3_c3 + r4_c1 + r4_c3 + r5_c1 + r5_c2 + r5_c3 == 0
r3_c2 + r3_c3 + r3_c4 + r4_c2 + r4_c4 + r5_c2 + r5_c3 + r5_c4 == 0
r3_c3 + r3_c4 + r3_c5 + r4_c3 + r4_c5 + r5_c3 + r5_c4 + r5_c5 == 0
r3_c4 + r3_c5 + r3_c6 + r4_c4 + r4_c6 + r5_c4 + r5_c5 + r5_c6 == 0
r3_c5 + r3_c6 + r3_c7 + r4_c5 + r4_c7 + r5_c5 + r5_c6 + r5_c7 == 0
r3_c6 + r3_c7 + r3_c8 + r4_c6 + r4_c8 + r5_c6 + r5_c7 + r5_c8 == 0
r3_c7 + r3_c8 + r3_c9 + r4_c7 + r4_c9 + r5_c7 + r5_c8 + r5_c9 == 0
r3_c8 + r3_c9 + r3_c10 + r4_c8 + r4_c10 + r5_c8 + r5_c9 + r5_c10 == 0
r4_c0 + r4_c1 + r4_c2 + r5_c0 + r5_c2 + r6_c0 + r6_c1 + r6_c2 == 1
r4_c1 + r4_c2 + r4_c3 + r5_c1 + r5_c3 + r6_c1 + r6_c2 + r6_c3 == 1
r4_c2 + r4_c3 + r4_c4 + r5_c2 + r5_c4 + r6_c2 + r6_c3 + r6_c4 == 1
r4_c3 + r4_c4 + r4_c5 + r5_c3 + r5_c5 + r6_c3 + r6_c4 + r6_c5 == 1
r4_c4 + r4_c5 + r4_c6 + r5_c4 + r5_c6 + r6_c4 + r6_c5 + r6_c6 == 1
r4_c5 + r4_c6 + r4_c7 + r5_c5 + r5_c7 + r6_c5 + r6_c6 + r6_c7 == 0
r4_c6 + r4_c7 + r4_c8 + r5_c6 + r5_c8 + r6_c6 + r6_c7 + r6_c8 == 0
r4_c7 + r4_c8 + r4_c9 + r5_c7 + r5_c9 + r6_c7 + r6_c8 + r6_c9 == 1
r4_c8 + r4_c9 + r4_c10 + r5_c8 + r5_c10 + r6_c8 + r6_c9 + r6_c10 == 1
r5_c4 + r5_c5 + r5_c6 + r6_c4 + r6_c6 + r7_c4 + r7_c5 + r7_c6 == 1
r5_c5 + r5_c6 + r5_c7 + r6_c5 + r6_c7 + r7_c5 + r7_c6 + r7_c7 == 0
r5_c6 + r5_c7 + r5_c8 + r6_c6 + r6_c8 + r7_c6 + r7_c7 + r7_c8 == 0
r5_c7 + r5_c8 + r5_c9 + r6_c7 + r6_c9 + r7_c7 + r7_c8 + r7_c9 == 1
r6_c4 + r6_c5 + r6_c6 + r7_c4 + r7_c6 + r8_c4 + r8_c5 + r8_c6 == 3
r6_c5 + r6_c6 + r6_c7 + r7_c5 + r7_c7 + r8_c5 + r8_c6 + r8_c7 == 1
r6_c6 + r6_c7 + r6_c8 + r7_c6 + r7_c8 + r8_c6 + r8_c7 + r8_c8 == 0
r6_c7 + r6_c8 + r6_c9 + r7_c7 + r7_c9 + r8_c7 + r8_c8 + r8_c9 == 1
r7_c5 + r7_c6 + r7_c7 + r8_c5 + r8_c7 + r9_c5 + r9_c6 + r9_c7 == 2
r7_c6 + r7_c7 + r7_c8 + r8_c6 + r8_c8 + r9_c6 + r9_c7 + r9_c8 == 1
r7_c7 + r7_c8 + r7_c9 + r8_c7 + r8_c9 + r9_c7 + r9_c8 + r9_c9 == 1
row=1 col=3, unsat!
```

My script tries to solve the equation with no luck (unsat). That means, no bomb can be at that cell.

Now let's run it for all cells:

```
row=1 col=3, unsat!
row=6 col=2, unsat!
row=6 col=3, unsat!
row=7 col=4, unsat!
row=7 col=9, unsat!
row=8 col=9, unsat!
```

These are cells where I can click safely, so I did:



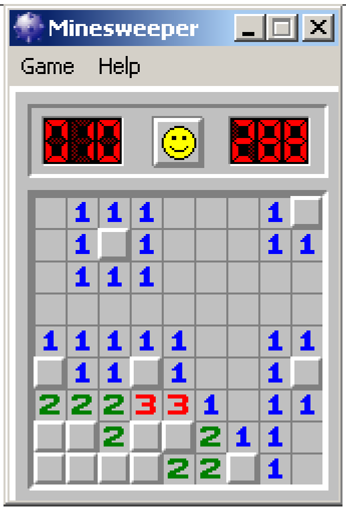
Now we have more information, so we update input:

```
known=[
"01110001?",
"01?100011",
"011100000",
"000000000",
"111110011",
"?11?1001?",
"???331011",
"?????2110",
"???????10"]
```

I run it again:

```
row=7 col=1, unsat!
row=7 col=2, unsat!
row=7 col=3, unsat!
row=8 col=3, unsat!
row=9 col=5, unsat!
row=9 col=6, unsat!
```

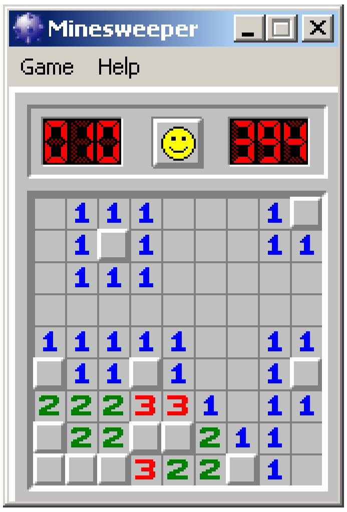
I click on these cells again:



I update it again:

```
known=[
"01110001?",
"01?100011",
"011100000",
"000000000",
"111110011",
"?11?1001?",
"222331011",
"??2??2110",
"????22?10"]
```

```
row=8 col=2, unsat!
row=9 col=4, unsat!
```



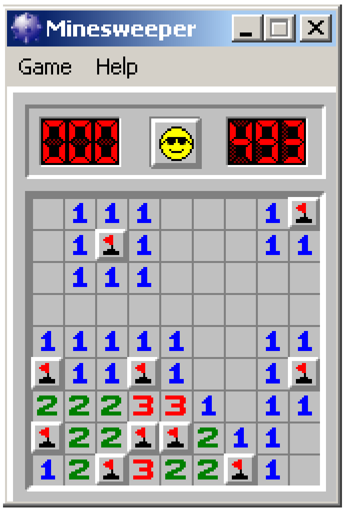
This is last update:

```
known=[
"01110001?",
"01?100011",
"011100000",
"000000000",
"111110011",
"?11?1001?",
"222331011",
"?22??2110",
"???322?10"]
```

...last result:

```
row=9 col=1, unsat!
row=9 col=2, unsat!
```

Hurrah!



The discussion at HN: https://news.ycombinator.com/item?id=13797375.

The files

https://smt.st/current_tree/equations/minesweeper/1_SMT.

3.10.2 Cracking Minesweeper with SAT solver

Simple *population count* function

First of all, somehow we need to count neighbour bombs. The counting function is similar to *population count* function.

We can try to make CNF expression using Wolfram Mathematica. This will be a function, returning *True* if any of 2 bits of 8 inputs bits are *True* and others are *False*. First, we make truth table of such function:

```
In[]:= tbl2 =
 Table[PadLeft[IntegerDigits[i, 2], 8] ->
   If[Equal[DigitCount[i, 2][[1]], 2], 1, 0], {i, 0, 255}]

Out[]= {{0, 0, 0, 0, 0, 0, 0, 0} -> 0, {0, 0, 0, 0, 0, 0, 0, 1} -> 0,
{0, 0, 0, 0, 0, 0, 1, 0} -> 0, {0, 0, 0, 0, 0, 0, 1, 1} -> 1,
{0, 0, 0, 0, 0, 1, 0, 0} -> 0, {0, 0, 0, 0, 0, 1, 0, 1} -> 1,
{0, 0, 0, 0, 0, 1, 1, 0} -> 1, {0, 0, 0, 0, 0, 1, 1, 1} -> 0,
{0, 0, 0, 0, 1, 0, 0, 0} -> 0, {0, 0, 0, 0, 1, 0, 0, 1} -> 1,
{0, 0, 0, 0, 1, 0, 1, 0} -> 1, {0, 0, 0, 0, 1, 0, 1, 1} -> 0,
...
{1, 1, 1, 1, 1, 0, 1, 0} -> 0, {1, 1, 1, 1, 1, 0, 1, 1} -> 0,
{1, 1, 1, 1, 1, 1, 0, 0} -> 0, {1, 1, 1, 1, 1, 1, 0, 1} -> 0,
{1, 1, 1, 1, 1, 1, 1, 0} -> 0, {1, 1, 1, 1, 1, 1, 1, 1} -> 0}
```

Now we can make CNF expression using this truth table:

```
In[]:= BooleanConvert[
 BooleanFunction[tbl2, {a, b, c, d, e, f, g, h}], "CNF"]

Out[]= (! a || ! b || ! c) && (! a || ! b || ! d) && (! a || !
    b || ! e) && (! a || ! b || ! f) && (! a || ! b || ! g) && (!
    a || ! b || ! h) && (! a || ! c || ! d) && (! a || ! c || !
    e) && (! a || ! c || ! f) && (! a || ! c || ! g) && (! a || !
```

```
  c || ! h) && (! a || ! d || ! e) && (! a || ! d || ! f) && (!
  a || ! d || ! g) && (! a || ! d || ! h) && (! a || ! e || !
  f) && (! a || ! e || ! g) && (! a || ! e || ! h) && (! a || !
  f || ! g) && (! a || ! f || ! h) && (! a || ! g || ! h) && (a ||
 b || c || d || e || f || g) && (a || b || c || d || e || f ||
 h) && (a || b || c || d || e || g || h) && (a || b || c || d || f ||
  g || h) && (a || b || c || e || f || g || h) && (a || b || d ||
 e || f || g || h) && (a || c || d || e || f || g ||
 h) && (! b || ! c || ! d) && (! b || ! c || ! e) && (! b || !
  c || ! f) && (! b || ! c || ! g) && (! b || ! c || ! h) && (!
  b || ! d || ! e) && (! b || ! d || ! f) && (! b || ! d || !
  g) && (! b || ! d || ! h) && (! b || ! e || ! f) && (! b || !
  e || ! g) && (! b || ! e || ! h) && (! b || ! f || ! g) && (!
  b || ! f || ! h) && (! b || ! g || ! h) && (b || c || d || e ||
 f || g ||
 h) && (! c || ! d || ! e) && (! c || ! d || ! f) && (! c || !
  d || ! g) && (! c || ! d || ! h) && (! c || ! e || ! f) && (!
  c || ! e || ! g) && (! c || ! e || ! h) && (! c || ! f || !
  g) && (! c || ! f || ! h) && (! c || ! g || ! h) && (! d || !
  e || ! f) && (! d || ! e || ! g) && (! d || ! e || ! h) && (!
  d || ! f || ! g) && (! d || ! f || ! h) && (! d || ! g || !
  h) && (! e || ! f || ! g) && (! e || ! f || ! h) && (! e || !
  g || ! h) && (! f || ! g || ! h)
```

The syntax is similar to C/C++. Let's check it.

I wrote a Python function to convert Mathematica's output into CNF file which can be feeded to SAT solver:

```
#!/usr/bin/python3

import subprocess

def mathematica_to_CNF (s, a):
    s=s.replace("a", a[0]).replace("b", a[1]).replace("c", a[2]).replace("d", a[3])
    s=s.replace("e", a[4]).replace("f", a[5]).replace("g", a[6]).replace("h", a[7])
    s=s.replace("!", "-").replace("||", " ").replace("(", "").replace(")", "")
    s=s.split ("&&")
    return s

def POPCNT2 (a):
    s="(!a||!b||!c)&&(!a||!b||!d)&&(!a||!b||!e)&&(!a||!b||!f)&&(!a||!b||!g)&&(!a||!b
        ||!h)&&(!a||!c||!d)&&" \
      "(!a||!c||!e)&&(!a||!c||!f)&&(!a||!c||!g)&&(!a||!c||!h)&&(!a||!d||!e)&&(!a||!d
        ||!f)&&(!a||!d||!g)&&" \
      "(!a||!d||!h)&&(!a||!e||!f)&&(!a||!e||!g)&&(!a||!e||!h)&&(!a||!f||!g)&&(!a||!f
        ||!h)&&(!a||!g||!h)&&" \
      "(a||b||c||d||e||f||g)&&(a||b||c||d||e||f||h)&&(a||b||c||d||e||g||h)&&(a||b||c
        ||d||f||g||h)&&" \
      "(a||b||c||e||f||g||h)&&(a||b||d||e||f||g||h)&&(a||c||d||e||f||g||h)&&(!b||!c
        ||!d)&&(!b||!c||!e)&&" \
      "(!b||!c||!f)&&(!b||!c||!g)&&(!b||!c||!h)&&(!b||!d||!e)&&(!b||!d||!f)&&(!b||!d
        ||!g)&&(!b||!d||!h)&&" \
      "(!b||!e||!f)&&(!b||!e||!g)&&(!b||!e||!h)&&(!b||!f||!g)&&(!b||!f||!h)&&(!b||!g
        ||!h)&&(b||c||d||e||f||g||h)&&" \
      "(!c||!d||!e)&&(!c||!d||!f)&&(!c||!d||!g)&&(!c||!d||!h)&&(!c||!e||!f)&&(!c||!e
        ||!g)&&(!c||!e||!h)&&" \
      "(!c||!f||!g)&&(!c||!f||!h)&&(!c||!g||!h)&&(!d||!e||!f)&&(!d||!e||!g)&&(!d||!e
        ||!h)&&(!d||!f||!g)&&" \
      "(!d||!f||!h)&&(!d||!g||!h)&&(!e||!f||!g)&&(!e||!f||!h)&&(!e||!g||!h)&&(!f||!g
        ||!h)"
    return mathematica_to_CNF(s, a)

clauses=POPCNT2(["1","2","3","4","5","6","7","8"])
```

```
f=open("tmp.cnf", "w")
f.write ("p cnf 8 "+str(len(clauses))+"\n")
for c in clauses:
    f.write(c+" 0\n")
f.close()
```

It replaces a/b/c/... variables to the variable names passed (1/2/3...), reworks syntax, etc. Here is a result:

```
p cnf 8 64
-1 -2 -3 0
-1 -2 -4 0
-1 -2 -5 0
-1 -2 -6 0
-1 -2 -7 0
-1 -2 -8 0
-1 -3 -4 0
-1 -3 -5 0
-1 -3 -6 0
-1 -3 -7 0
-1 -3 -8 0
-1 -4 -5 0
-1 -4 -6 0
-1 -4 -7 0
-1 -4 -8 0
-1 -5 -6 0
-1 -5 -7 0
-1 -5 -8 0
-1 -6 -7 0
-1 -6 -8 0
-1 -7 -8 0
1 2 3 4 5 6 7 0
1 2 3 4 5 6 8 0
1 2 3 4 5 7 8 0
1 2 3 4 6 7 8 0
1 2 3 5 6 7 8 0
1 2 4 5 6 7 8 0
1 3 4 5 6 7 8 0
-2 -3 -4 0
-2 -3 -5 0
-2 -3 -6 0
-2 -3 -7 0
-2 -3 -8 0
-2 -4 -5 0
-2 -4 -6 0
-2 -4 -7 0
-2 -4 -8 0
-2 -5 -6 0
-2 -5 -7 0
-2 -5 -8 0
-2 -6 -7 0
-2 -6 -8 0
-2 -7 -8 0
2 3 4 5 6 7 8 0
-3 -4 -5 0
-3 -4 -6 0
-3 -4 -7 0
-3 -4 -8 0
-3 -5 -6 0
-3 -5 -7 0
-3 -5 -8 0
-3 -6 -7 0
```

```
-3 -6 -8 0
-3 -7 -8 0
-4 -5 -6 0
-4 -5 -7 0
-4 -5 -8 0
-4 -6 -7 0
-4 -6 -8 0
-4 -7 -8 0
-5 -6 -7 0
-5 -6 -8 0
-5 -7 -8 0
-6 -7 -8 0
```

I can run it:

```
% minisat -verb=0 tst1.cnf results.txt
SATISFIABLE

% cat results.txt
SAT
1 -2 -3 -4 -5 -6 -7 8 0
```

The variable name in results lacking minus sign is *True*. Variable name with minus sign is *False*. We see there are just two variables are *True*: 1 and 8. This is indeed correct: MiniSat solver found a condition, for which our function returns *True*. Zero at the end is just a terminal symbol which means nothing.

We can ask MiniSat for another solution, by adding current solution to the input CNF file, but with all variables negated:

```
...
-5 -6 -8 0
-5 -7 -8 0
-6 -7 -8 0
-1 2 3 4 5 6 7 -8 0
```

In plain English language, this means "give me ANY solution which can satisfy all clauses, but also not equal to the last clause we've just added".

MiniSat, indeed, found another solution, again, with only 2 variables equal to *True*:

```
% minisat -verb=0 tst2.cnf results.txt
SATISFIABLE

% cat results.txt
SAT
1 2 -3 -4 -5 -6 -7 -8 0
```

By the way, *population count* function for 8 neighbours (POPCNT8) in CNF form is simplest:

```
a&&b&&c&&d&&e&&f&&g&&h
```

Indeed: it's true if all 8 input bits are *True*.

The function for 0 neighbours (POPCNT0) is also simple:

```
!a&&!b&&!c&&!d&&!e&&!f&&!g&&!h
```

It means, it will return *True*, if all input variables are *False*.

By the way, POPCNT1 function is also simple:

```
(!a||!b)&&(!a||!c)&&(!a||!d)&&(!a||!e)&&(!a||!f)&&(!a||!g)&&(!a||!h)&&(a||b||c||d||e
    ||f||g||h)&&
(!b||!c)&&(!b||!d)&&(!b||!e)&&(!b||!f)&&(!b||!g)&&(!b||!h)&&(!c||!d)&&(!c||!e)&&(!c
    ||!f)&&(!c||!g)&&
(!c||!h)&&(!d||!e)&&(!d||!f)&&(!d||!g)&&(!d||!h)&&(!e||!f)&&(!e||!g)&&(!e||!h)&&(!f
    ||!g)&&(!f||!h)&&(!g||!h)
```

There is just enumeration of all possible pairs of 8 variables (a/b, a/c, a/d, etc), which implies: no two bits must be present simultaneously in each possible pair. And there is another clause: "(a||b||c||d||e||f||g||h)", which implies: at least one bit must be present among 8 variables.

And yes, you can ask Mathematica for finding CNF expressions for any other truth table.

Minesweeper

Now we can use Mathematica to generate all *population count* functions for 0..8 neighbours.

For 9·9 Minesweeper matrix including invisible border, there will be 11·11 = 121 variables, mapped to Minesweeper matrix like this:

```
 1    2   3   4   5   6   7   8   9  10  11
12   13  14  15  16  17  18  19  20  21  22
23   24  25  26  27  28  29  30  31  32  33
34   35  36  37  38  39  40  41  42  43  44

...

100 101 102 103 104 105 106 107 108 109 110
111 112 113 114 115 116 117 118 119 120 121
```

Then we write a Python script which stacks all *population count* functions: each function for each known number of neighbours (digit on Minesweeper field). Each POPCNTx() function takes list of variable numbers and outputs list of clauses to be added to the final CNF file.

As of empty cells, we also add them as clauses, but with minus sign, which means, the variable must be *False*. Whenever we try to place bomb, we add its variable as clause without minus sign, this means the variable must be *True*.

Then we execute external minisat process. The only thing we need from it is exit code. If an input CNF is `UNSAT`, it returns 20:

We use here the information from the previous solving of Minesweeper: 3.10.1.

```
#!/usr/bin/python3

import subprocess

WIDTH=9
HEIGHT=9
VARS_TOTAL=(WIDTH+2)*(HEIGHT+2)

known=[
"01?10001?",
"01?100011",
"011100000",
"000000000",
"111110011",
"????1001?",
"????3101?",
"?????211?",
"?????????"]

def mathematica_to_CNF (s, a):
    s=s.replace("a", a[0]).replace("b", a[1]).replace("c", a[2]).replace("d", a[3])
    s=s.replace("e", a[4]).replace("f", a[5]).replace("g", a[6]).replace("h", a[7])
    s=s.replace("!", "-").replace("||", " ").replace("(", "").replace(")", "")
    s=s.split ("&&")
    return s

def POPCNT0 (a):
    s="!a&&!b&&!c&&!d&&!e&&!f&&!g&&!h"
    return mathematica_to_CNF(s, a)

def POPCNT1 (a):
    s="(!a||!b)&&(!a||!c)&&(!a||!d)&&(!a||!e)&&(!a||!f)&&(!a||!g)&&(!a||!h)&&(a||b||c
       ||d||e||f||g||h)&&" \
      "(!b||!c)&&(!b||!d)&&(!b||!e)&&(!b||!f)&&(!b||!g)&&(!b||!h)&&(!c||!d)&&(!c||!e)
         &&(!c||!f)&&(!c||!g)&&" \
      "(!c||!h)&&(!d||!e)&&(!d||!f)&&(!d||!g)&&(!d||!h)&&(!e||!f)&&(!e||!g)&&(!e||!h)
         &&(!f||!g)&&(!f||!h)&&(!g||!h)"
    return mathematica_to_CNF(s, a)

def POPCNT2 (a):
    s="(!a||!b||!c)&&(!a||!b||!d)&&(!a||!b||!e)&&(!a||!b||!f)&&(!a||!b||!g)&&(!a||!b
       ||!h)&&(!a||!c||!d)&&" \
      "(!a||!c||!e)&&(!a||!c||!f)&&(!a||!c||!g)&&(!a||!c||!h)&&(!a||!d||!e)&&(!a||!d
         ||!f)&&(!a||!d||!g)&&" \
      "(!a||!d||!h)&&(!a||!e||!f)&&(!a||!e||!g)&&(!a||!e||!h)&&(!a||!f||!g)&&(!a||!f
         ||!h)&&(!a||!g||!h)&&" \
      "(a||b||c||d||e||f||g)&&(a||b||c||d||e||f||h)&&(a||b||c||d||e||g||h)&&(a||b||c
         ||d||f||g||h)&&" \
      "(a||b||c||e||f||g||h)&&(a||b||d||e||f||g||h)&&(a||c||d||e||f||g||h)&&(!b||!c
         ||!d)&&(!b||!c||!e)&&" \
      "(!b||!c||!f)&&(!b||!c||!g)&&(!b||!c||!h)&&(!b||!d||!e)&&(!b||!d||!f)&&(!b||!d
         ||!g)&&(!b||!d||!h)&&" \
      "(!b||!e||!f)&&(!b||!e||!g)&&(!b||!e||!h)&&(!b||!f||!g)&&(!b||!f||!h)&&(!b||!g
         ||!h)&&(b||c||d||e||f||g||h)&&" \
      "(!c||!d||!e)&&(!c||!d||!f)&&(!c||!d||!g)&&(!c||!d||!h)&&(!c||!e||!f)&&(!c||!e
         ||!g)&&(!c||!e||!h)&&" \
      "(!c||!f||!g)&&(!c||!f||!h)&&(!c||!g||!h)&&(!d||!e||!f)&&(!d||!e||!g)&&(!d||!e
         ||!h)&&(!d||!f||!g)&&" \
      "(!d||!f||!h)&&(!d||!g||!h)&&(!e||!f||!g)&&(!e||!f||!h)&&(!e||!g||!h)&&(!f||!g
         ||!h)"
    return mathematica_to_CNF(s, a)

def POPCNT3 (a):
    s="(!a||!b||!c||!d)&&(!a||!b||!c||!e)&&(!a||!b||!c||!f)&&(!a||!b||!c||!g)&&(!a||!
       b||!c||!h)&&" \
      "(!a||!b||!d||!e)&&(!a||!b||!d||!f)&&(!a||!b||!d||!g)&&(!a||!b||!d||!h)&&(!a||!
         b||!e||!f)&&" \
      "(!a||!b||!e||!g)&&(!a||!b||!e||!h)&&(!a||!b||!f||!g)&&(!a||!b||!f||!h)&&(!a||!
         b||!g||!h)&&" \
      "(!a||!c||!d||!e)&&(!a||!c||!d||!f)&&(!a||!c||!d||!g)&&(!a||!c||!d||!h)&&(!a||!
         c||!e||!f)&&" \
      "(!a||!c||!e||!g)&&(!a||!c||!e||!h)&&(!a||!c||!f||!g)&&(!a||!c||!f||!h)&&(!a||!
         c||!g||!h)&&" \
      "(!a||!d||!e||!f)&&(!a||!d||!e||!g)&&(!a||!d||!e||!h)&&(!a||!d||!f||!g)&&(!a||!
         d||!f||!h)&&" \
      "(!a||!d||!g||!h)&&(!a||!e||!f||!g)&&(!a||!e||!f||!h)&&(!a||!e||!g||!h)&&(!a||!
         f||!g||!h)&&" \
      "(a||b||c||d||e||f)&&(a||b||c||d||e||g)&&(a||b||c||d||e||h)&&(a||b||c||d||f||g)
         &&(a||b||c||d||f||h)&&" \
      "(a||b||c||d||g||h)&&(a||b||c||e||f||g)&&(a||b||c||e||f||h)&&(a||b||c||e||g||h)
         &&(a||b||c||f||g||h)&&" \
      "(a||b||d||e||f||g)&&(a||b||d||e||f||h)&&(a||b||d||e||g||h)&&(a||b||d||f||g||h)
         &&(a||b||e||f||g||h)&&" \
      "(a||c||d||e||f||g)&&(a||c||d||e||f||h)&&(a||c||d||e||g||h)&&(a||c||d||f||g||h)
         &&(a||c||e||f||g||h)&&" \
      "(a||d||e||f||g||h)&&(!b||!c||!d||!e)&&(!b||!c||!d||!f)&&(!b||!c||!d||!g)&&(!b
         ||!c||!d||!h)&&" \
      "(!b||!c||!e||!f)&&(!b||!c||!e||!g)&&(!b||!c||!e||!h)&&(!b||!c||!f||!g)&&(!b||!
         c||!f||!h)&&" \
      "(!b||!c||!g||!h)&&(!b||!d||!e||!f)&&(!b||!d||!e||!g)&&(!b||!d||!e||!h)&&(!b||!
         d||!f||!g)&&" \
      "(!b||!d||!f||!h)&&(!b||!d||!g||!h)&&(!b||!e||!f||!g)&&(!b||!e||!f||!h)&&(!b||!
        e||!g||!h)&&" \
      "(!b||!f||!g||!h)&&(b||c||d||e||f||g)&&(b||c||d||e||f||h)&&(b||c||d||e||g||h)
        &&(b||c||d||f||g||h)&&" \
      "(b||c||e||f||g||h)&&(b||d||e||f||g||h)&&(!c||!d||!e||!f)&&(!c||!d||!e||!g)&&(!
        c||!d||!e||!h)&&" \
      "(!c||!d||!f||!g)&&(!c||!d||!f||!h)&&(!c||!d||!g||!h)&&(!c||!e||!f||!g)&&(!c||!
        e||!f||!h)&&" \
      "(!c||!e||!g||!h)&&(!c||!f||!g||!h)&&(c||d||e||f||g||h)&&(!d||!e||!f||!g)&&(!d
        ||!e||!f||!h)&&" \
      "(!d||!e||!g||!h)&&(!d||!f||!g||!h)&&(!e||!f||!g||!h)"
    return mathematica_to_CNF(s, a)

def POPCNT4 (a):
    s="(!a||!b||!c||!d||!e)&&(!a||!b||!c||!d||!f)&&(!a||!b||!c||!d||!g)&&(!a||!b||!c
       ||!d||!h)&&" \
      "(!a||!b||!c||!e||!f)&&(!a||!b||!c||!e||!g)&&(!a||!b||!c||!e||!h)&&(!a||!b||!c
        ||!f||!g)&&" \
      "(!a||!b||!c||!f||!h)&&(!a||!b||!c||!g||!h)&&(!a||!b||!d||!e||!f)&&(!a||!b||!d
        ||!e||!g)&&" \
      "(!a||!b||!d||!e||!h)&&(!a||!b||!d||!f||!g)&&(!a||!b||!d||!f||!h)&&(!a||!b||!d
        ||!g||!h)&&" \
      "(!a||!b||!e||!f||!g)&&(!a||!b||!e||!f||!h)&&(!a||!b||!e||!g||!h)&&(!a||!b||!f
        ||!g||!h)&&" \
      "(!a||!c||!d||!e||!f)&&(!a||!c||!d||!e||!g)&&(!a||!c||!d||!e||!h)&&(!a||!c||!d
        ||!f||!g)&&" \
      "(!a||!c||!d||!f||!h)&&(!a||!c||!d||!g||!h)&&(!a||!c||!e||!f||!g)&&(!a||!c||!e
        ||!f||!h)&&" \
      "(!a||!c||!e||!g||!h)&&(!a||!c||!f||!g||!h)&&(!a||!d||!e||!f||!g)&&(!a||!d||!e
        ||!f||!h)&&" \
      "(!a||!d||!e||!g||!h)&&(!a||!d||!f||!g||!h)&&(!a||!e||!f||!g||!h)&&(a||b||c||d
        ||e)&&(a||b||c||d||f)&&" \
      "(a||b||c||d||g)&&(a||b||c||d||h)&&(a||b||c||e||f)&&(a||b||c||e||g)&&(a||b||c||
        e||h)&&(a||b||c||f||g)&&" \
      "(a||b||c||f||h)&&(a||b||c||g||h)&&(a||b||d||e||f)&&(a||b||d||e||g)&&(a||b||d||
        e||h)&&(a||b||d||f||g)&&" \
      "(a||b||d||f||h)&&(a||b||d||g||h)&&(a||b||e||f||g)&&(a||b||e||f||h)&&(a||b||e||
        g||h)&&(a||b||f||g||h)&&" \
      "(a||c||d||e||f)&&(a||c||d||e||g)&&(a||c||d||e||h)&&(a||c||d||f||g)&&(a||c||d||
        f||h)&&(a||c||d||g||h)&&" \
      "(a||c||e||f||g)&&(a||c||e||f||h)&&(a||c||e||g||h)&&(a||c||f||g||h)&&(a||d||e||
        f||g)&&(a||d||e||f||h)&&" \
      "(a||d||e||g||h)&&(a||d||f||g||h)&&(a||e||f||g||h)&&(!b||!c||!d||!e||!f)&&(!b
        ||!c||!d||!e||!g)&&" \
      "(!b||!c||!d||!e||!h)&&(!b||!c||!d||!f||!g)&&(!b||!c||!d||!f||!h)&&(!b||!c||!d
        ||!g||!h)&&" \
      "(!b||!c||!e||!f||!g)&&(!b||!c||!e||!f||!h)&&(!b||!c||!e||!g||!h)&&(!b||!c||!f
        ||!g||!h)&&" \
      "(!b||!d||!e||!f||!g)&&(!b||!d||!e||!f||!h)&&(!b||!d||!e||!g||!h)&&(!b||!d||!f
        ||!g||!h)&&" \
      "(!b||!e||!f||!g||!h)&&(b||c||d||e||f)&&(b||c||d||e||g)&&(b||c||d||e||h)&&(b||c
        ||d||f||g)&&" \
      "(b||c||d||f||h)&&(b||c||d||g||h)&&(b||c||e||f||g)&&(b||c||e||f||h)&&(b||c||e||
        g||h)&&" \
      "(b||c||f||g||h)&&(b||d||e||f||g)&&(b||d||e||f||h)&&(b||d||e||g||h)&&(b||d||f||
        g||h)&&" \
      "(b||e||f||g||h)&&(!c||!d||!e||!f||!g)&&(!c||!d||!e||!f||!h)&&(!c||!d||!e||!g
        ||!h)&&" \
      "(!c||!d||!f||!g||!h)&&(!c||!e||!f||!g||!h)&&(c||d||e||f||g)&&(c||d||e||f||h)
        &&(c||d||e||g||h)&&" \
      "(c||d||f||g||h)&&(c||e||f||g||h)&&(!d||!e||!f||!g||!h)&&(d||e||f||g||h)"
    return mathematica_to_CNF(s, a)
```

```
def POPCNT5 (a):
    s="(!a||!b||!c||!d||!e||!f)&&(!a||!b||!c||!d||!e||!g)&&(!a||!b||!c||!d||!e||!h)&&" \
      "(!a||!b||!c||!d||!f||!g)&&(!a||!b||!c||!d||!f||!h)&&(!a||!b||!c||!d||!g||!h)&&" \
      "(!a||!b||!c||!e||!f||!g)&&(!a||!b||!c||!e||!f||!h)&&(!a||!b||!c||!e||!g||!h)&&" \
      "(!a||!b||!c||!f||!g||!h)&&(!a||!b||!d||!e||!f||!g)&&(!a||!b||!d||!e||!f||!h)&&" \
      "(!a||!b||!d||!e||!g||!h)&&(!a||!b||!d||!f||!g||!h)&&(!a||!b||!e||!f||!g||!h)&&" \
      "(!a||!c||!d||!e||!f||!g)&&(!a||!c||!d||!e||!f||!h)&&(!a||!c||!d||!e||!g||!h)&&" \
      "(!a||!c||!d||!f||!g||!h)&&(!a||!c||!e||!f||!g||!h)&&(!a||!d||!e||!f||!g||!h)&&" \
      "(a||b||c||d)&&(a||b||c||e)&&(a||b||c||f)&&(a||b||c||g)&&(a||b||c||h)&&(a||b||d||e)&&" \
      "(a||b||d||f)&&(a||b||d||g)&&(a||b||d||h)&&(a||b||e||f)&&(a||b||e||g)&&(a||b||e||h)&&" \
      "(a||b||f||g)&&(a||b||f||h)&&(a||b||g||h)&&(a||c||d||e)&&(a||c||d||f)&&(a||c||d||g)&&" \
      "(a||c||d||h)&&(a||c||e||f)&&(a||c||e||g)&&(a||c||e||h)&&(a||c||f||g)&&(a||c||f||h)&&" \
      "(a||c||g||h)&&(a||d||e||f)&&(a||d||e||g)&&(a||d||e||h)&&(a||d||f||g)&&(a||d||f||h)&&" \
      "(a||d||g||h)&&(a||e||f||g)&&(a||e||f||h)&&(a||e||g||h)&&(a||f||g||h)&&(!b||!c||!d||!e||!f||!g)&&" \
      "(!b||!c||!d||!e||!f||!h)&&(!b||!c||!d||!e||!g||!h)&&(!b||!c||!d||!f||!g||!h)&&" \
      "(!b||!c||!e||!f||!g||!h)&&(!b||!d||!e||!f||!g||!h)&&(b||c||d||e)&&(b||c||d||f)&&" \
      "(b||c||d||g)&&(b||c||d||h)&&(b||c||e||f)&&(b||c||e||g)&&(b||c||e||h)&&(b||c||f||g)&&" \
      "(b||c||f||h)&&(b||c||g||h)&&(b||d||e||f)&&(b||d||e||g)&&(b||d||e||h)&&(b||d||f||g)&&" \
      "(b||d||f||h)&&(b||d||g||h)&&(b||e||f||g)&&(b||e||f||h)&&(b||e||g||h)&&(b||f||g||h)&&" \
      "(!c||!d||!e||!f||!g||!h)&&(c||d||e||f)&&(c||d||e||g)&&(c||d||e||h)&&(c||d||f||g)&&" \
      "(c||d||f||h)&&(c||d||g||h)&&(c||e||f||g)&&(c||e||f||h)&&(c||e||g||h)&&(c||f||g||h)&&" \
      "(d||e||f||g)&&(d||e||f||h)&&(d||e||g||h)&&(d||f||g||h)&&(e||f||g||h)"
    return mathematica_to_CNF(s, a)

def POPCNT6 (a):
    s="(!a||!b||!c||!d||!e||!f||!g)&&(!a||!b||!c||!d||!e||!f||!h)&&(!a||!b||!c||!d||!e||!g||!h)&&" \
      "(!a||!b||!c||!d||!f||!g||!h)&&(!a||!b||!c||!e||!f||!g||!h)&&(!a||!b||!d||!e||!f||!g||!h)&&" \
      "(!a||!c||!d||!e||!f||!g||!h)&&(a||b||c)&&(a||b||d)&&(a||b||e)&&(a||b||f)&&(a||b||g)&&(a||b||h)&&" \
      "(a||c||d)&&(a||c||e)&&(a||c||f)&&(a||c||g)&&(a||c||h)&&(a||d||e)&&(a||d||f)&&(a||d||g)&&" \
      "(a||d||h)&&(a||e||f)&&(a||e||g)&&(a||e||h)&&(a||f||g)&&(a||f||h)&&(a||g||h)&&" \
      "(!b||!c||!d||!e||!f||!g||!h)&&(b||c||d)&&(b||c||e)&&(b||c||f)&&(b||c||g)&&(b||c||h)&&(b||d||e)&&" \
      "(b||d||f)&&(b||d||g)&&(b||d||h)&&(b||e||f)&&(b||e||g)&&(b||e||h)&&(b||f||g)&&(b||f||h)&&(b||g||h)&&" \
      "(c||d||e)&&(c||d||f)&&(c||d||g)&&(c||d||h)&&(c||e||f)&&(c||e||g)&&(c||e||h)&&(
         c||f||g)&&(c||f||h)&&" \
      "(c||g||h)&&(d||e||f)&&(d||e||g)&&(d||e||h)&&(d||f||g)&&(d||f||h)&&(d||g||h)&&"
          \
      "(e||f||g)&&(e||f||h)&&(e||g||h)&&(f||g||h)"
    return mathematica_to_CNF(s, a)

def POPCNT7 (a):
    s="(!a||!b||!c||!d||!e||!f||!g||!h)&&(a||b)&&(a||c)&&(a||d)&&(a||e)&&(a||f)&&(a||
       g)&&(a||h)&&(b||c)&&" \
      "(b||d)&&(b||e)&&(b||f)&&(b||g)&&(b||h)&&(c||d)&&(c||e)&&(c||f)&&(c||g)&&(c||h)
         &&(d||e)&&(d||f)&&(d||g)&&" \
      "(d||h)&&(e||f)&&(e||g)&&(e||h)&&(f||g)&&(f||h)&&(g||h)"
    return mathematica_to_CNF(s, a)

def POPCNT8 (a):
    s="a&&b&&c&&d&&e&&f&&g&&h"
    return mathematica_to_CNF(s, a)

POPCNT_functions=[POPCNT0, POPCNT1, POPCNT2, POPCNT3, POPCNT4, POPCNT5, POPCNT6,
   POPCNT7, POPCNT8]

def coords_to_var (row, col):
    # we always use SAT variables as strings, anyway.
    # the 1st variables is 1, not 0
    return str(row*(WIDTH+2)+col+1)

def chk_bomb(row, col):
    clauses=[]

    # make empty border
    # all variables are negated (because they must be False)
    for c in range(WIDTH+2):
        clauses.append ("-"+coords_to_var(0,c))
        clauses.append ("-"+coords_to_var(HEIGHT+1,c))
    for r in range(HEIGHT+2):
        clauses.append ("-"+coords_to_var(r,0))
        clauses.append ("-"+coords_to_var(r,WIDTH+1))

    for r in range(1,HEIGHT+1):
        for c in range(1,WIDTH+1):
            t=known[r-1][c-1]
            if t in "012345678":
                # cell at r, c is empty (False):
                clauses.append ("-"+coords_to_var(r,c))
                # we need an empty border so the following expression would work for
                    all possible cells:
                neighbours=[coords_to_var(r-1, c-1), coords_to_var(r-1, c),
                    coords_to_var(r-1, c+1), coords_to_var(r, c-1),
                          coords_to_var(r, c+1), coords_to_var(r+1, c-1), coords_to_var
                              (r+1, c), coords_to_var(r+1, c+1)]
                clauses=clauses+POPCNT_functions[int(t)](neighbours)

    # place a bomb
    clauses.append (coords_to_var(row,col))

    f=open("tmp.cnf", "w")
    f.write ("p cnf "+str(VARS_TOTAL)+" "+str(len(clauses))+"\n")
    for c in clauses:
        f.write(c+" 0\n")
    f.close()
    child = subprocess.Popen(["minisat", "tmp.cnf"], stdout=subprocess.PIPE)
    child.wait()
    # 10 is SAT, 20 is UNSAT
    if child.returncode==20:
        print ("row=%d, col=%d, unsat!" % (row, col))

for r in range(1,HEIGHT+1):
    for c in range(1,WIDTH+1):
        if known[r-1][c-1]=="?":
            chk_bomb(r, c)
```

(https://smt.st/current_tree/equations/minesweeper/2_SAT/minesweeper_SAT.py)

The output CNF file can be large, up to ≈ 2000 clauses, or more, here is an example: https://smt.st/current_tree/equations/minesweeper/2_SAT/sample.cnf.

Anyway, it works just like my previous Z3Py script:

```
row=1, col=3, unsat!
row=6, col=2, unsat!
row=6, col=3, unsat!
row=7, col=4, unsat!
row=7, col=9, unsat!
row=8, col=9, unsat!
```

...but it runs way faster, even considering overhead of executing external program. Perhaps, Z3Py version could be optimized better?

The files, including Wolfram Mathematica notebook: https://smt.st/current_tree/equations/minesweeper/2_SAT.

3.10.3 Optimization

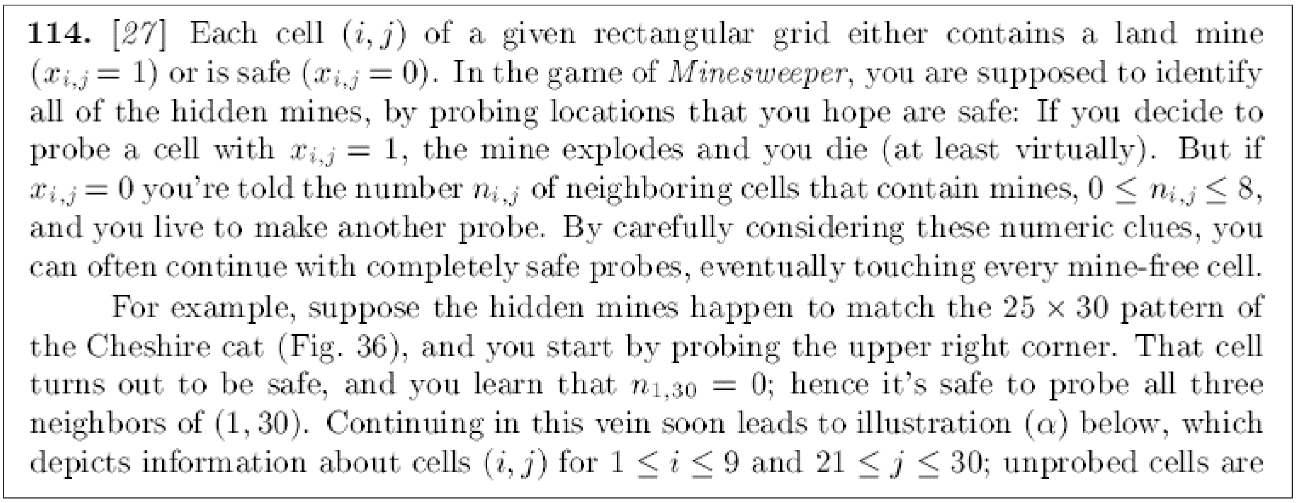
A simple observation can help us: there is no need to probe hidden cells in the middle of other hidden cells. In fact, only those hidden cells surrounding digits (known number of bombs) is to be probed: It speeds up everything a little.

```
...
    def get_from_known_or_empty(known, r, c):
        # I'm lazy to do overflow checks
        try:
            return known[r][c]
        except IndexError:
            return ""

    def have_neighbour_digit(known, r, c):
        # returns True if neighbour is digit, False otherwise
        digits="0123456789"
        t=[]
        t.append(get_from_known_or_empty(known, r-1, c-1) in digits)
        t.append(get_from_known_or_empty(known, r-1, c) in digits)
        t.append(get_from_known_or_empty(known, r-1, c+1) in digits)
        t.append(get_from_known_or_empty(known, r, c-1) in digits)
        t.append(get_from_known_or_empty(known, r, c+1) in digits)
        t.append(get_from_known_or_empty(known, r+1, c-1) in digits)
        t.append(get_from_known_or_empty(known, r+1, c) in digits)
        t.append(get_from_known_or_empty(known, r+1, c+1) in digits)
        return any(t)

    # enumerate all hidden cells:
    rt=[]
    for r in range(1,HEIGHT+1):
        for c in range(1,WIDTH+1):
            if known[r-1][c-1]=="?" and have_neighbour_digit(known, r-1, c-1):
                #print "checking", r, c
                rt.append(chk_bomb(r, c))
    return filter(None, rt)
...
```

(https://smt.st/current_tree/equations/minesweeper/3_SMT_opt/minesweeper_Z3_with_tests.py)

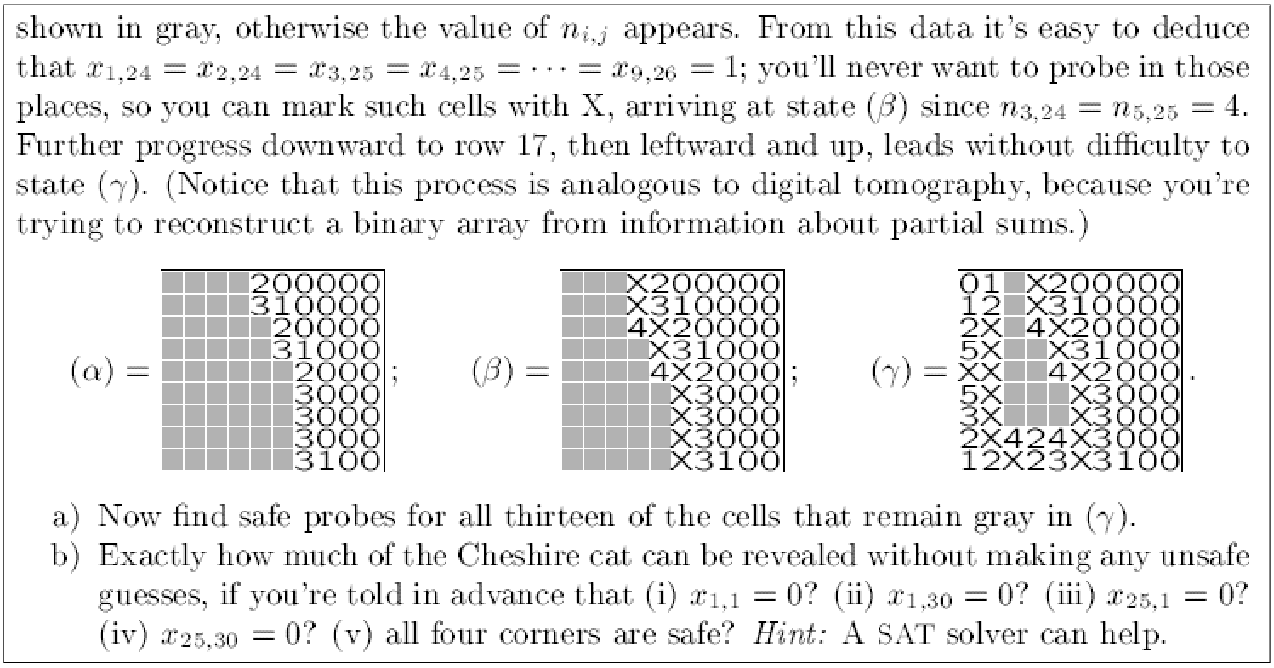
3.10.4 Cracking Minesweeper: Donald Knuth's version

From Donald Knuth's TAOCP[5] Section 7.2.2.2:



114. [*27*] Each cell (i, j) of a given rectangular grid either contains a land mine ($x_{i,j} = 1$) or is safe ($x_{i,j} = 0$). In the game of *Minesweeper*, you are supposed to identify all of the hidden mines, by probing locations that you hope are safe: If you decide to probe a cell with $x_{i,j} = 1$, the mine explodes and you die (at least virtually). But if $x_{i,j} = 0$ you're told the number $n_{i,j}$ of neighboring cells that contain mines, $0 \le n_{i,j} \le 8$, and you live to make another probe. By carefully considering these numeric clues, you can often continue with completely safe probes, eventually touching every mine-free cell.

For example, suppose the hidden mines happen to match the 25 × 30 pattern of the Cheshire cat (Fig. 36), and you start by probing the upper right corner. That cell turns out to be safe, and you learn that $n_{1,30} = 0$; hence it's safe to probe all three neighbors of $(1, 30)$. Continuing in this vein soon leads to illustration (α) below, which depicts information about cells (i, j) for $1 \le i \le 9$ and $21 \le j \le 30$; unprobed cells are



shown in gray, otherwise the value of $n_{i,j}$ appears. From this data it's easy to deduce that $x_{1,24} = x_{2,24} = x_{3,25} = x_{4,25} = \cdots = x_{9,26} = 1$; you'll never want to probe in those places, so you can mark such cells with X, arriving at state (β) since $n_{3,24} = n_{5,25} = 4$. Further progress downward to row 17, then leftward and up, leads without difficulty to state (γ). (Notice that this process is analogous to digital tomography, because you're trying to reconstruct a binary array from information about partial sums.)

$(\alpha) =$

```
▒▒▒▒200000
▒▒▒▒310000
▒▒▒▒▒20000
▒▒▒▒▒31000
▒▒▒▒▒▒2000
▒▒▒▒▒▒3000
▒▒▒▒▒▒3000
▒▒▒▒▒▒3000
▒▒▒▒▒▒3100
```

;

$(\beta) =$

```
▒▒▒X200000
▒▒▒X310000
▒▒▒4X20000
▒▒▒▒X31000
▒▒▒▒4X2000
▒▒▒▒▒X3000
▒▒▒▒▒X3000
▒▒▒▒▒X3000
▒▒▒▒▒X3100
```

;

$(\gamma) =$

```
01▒X200000
12▒X310000
2X▒4X20000
5X▒▒X31000
XX▒▒4X2000
5X▒▒▒X3000
3X▒▒▒X3000
2X424X3000
12X23X3100
```

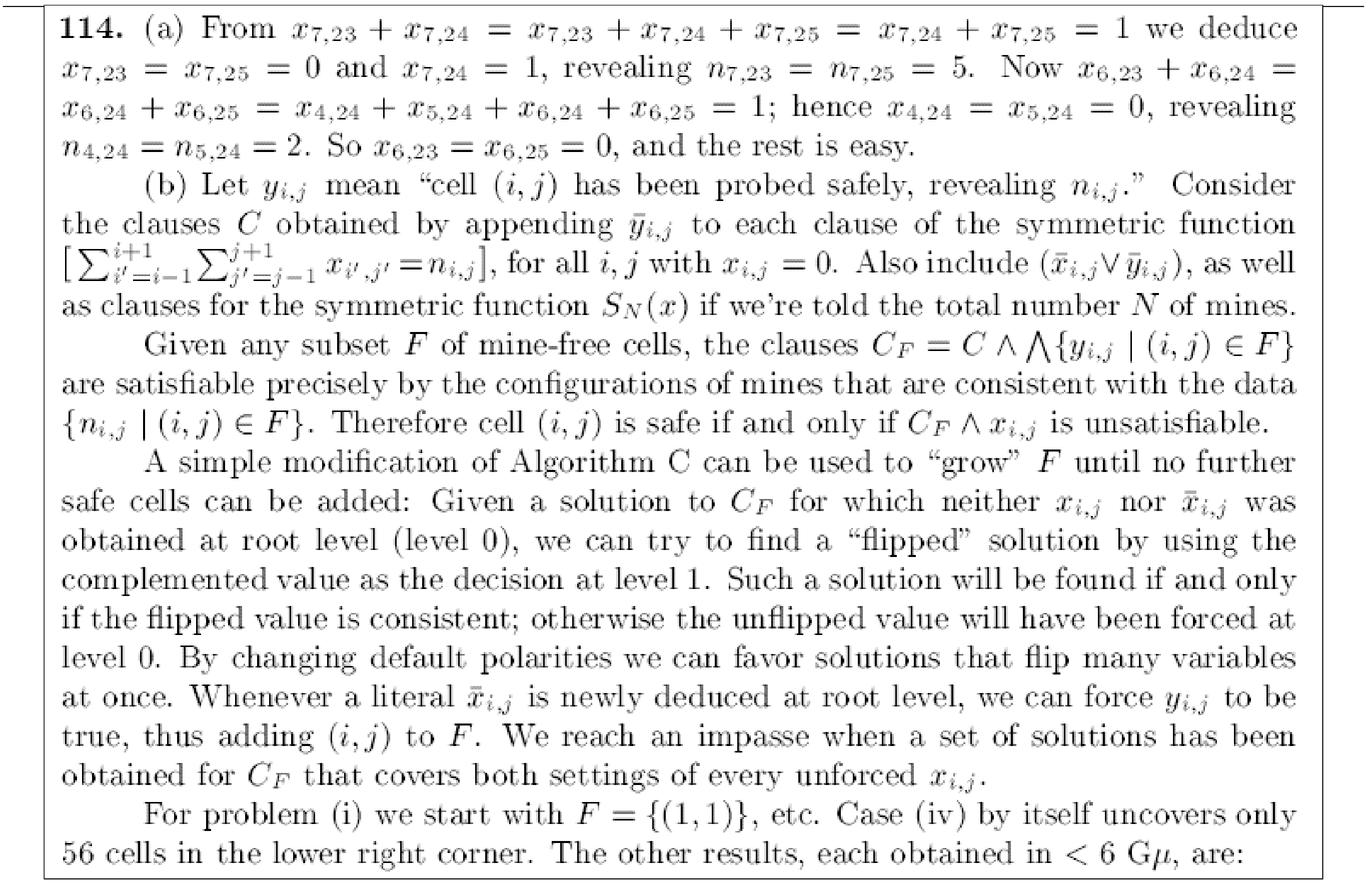
.

a) Now find safe probes for all thirteen of the cells that remain gray in (γ).

b) Exactly how much of the Cheshire cat can be revealed without making any unsafe guesses, if you're told in advance that (i) $x_{1,1} = 0$? (ii) $x_{1,30} = 0$? (iii) $x_{25,1} = 0$? (iv) $x_{25,30} = 0$? (v) all four corners are safe? *Hint:* A SAT solver can help.

And the solution:

[5] The Art Of Computer Programming (Donald Knuth's book)



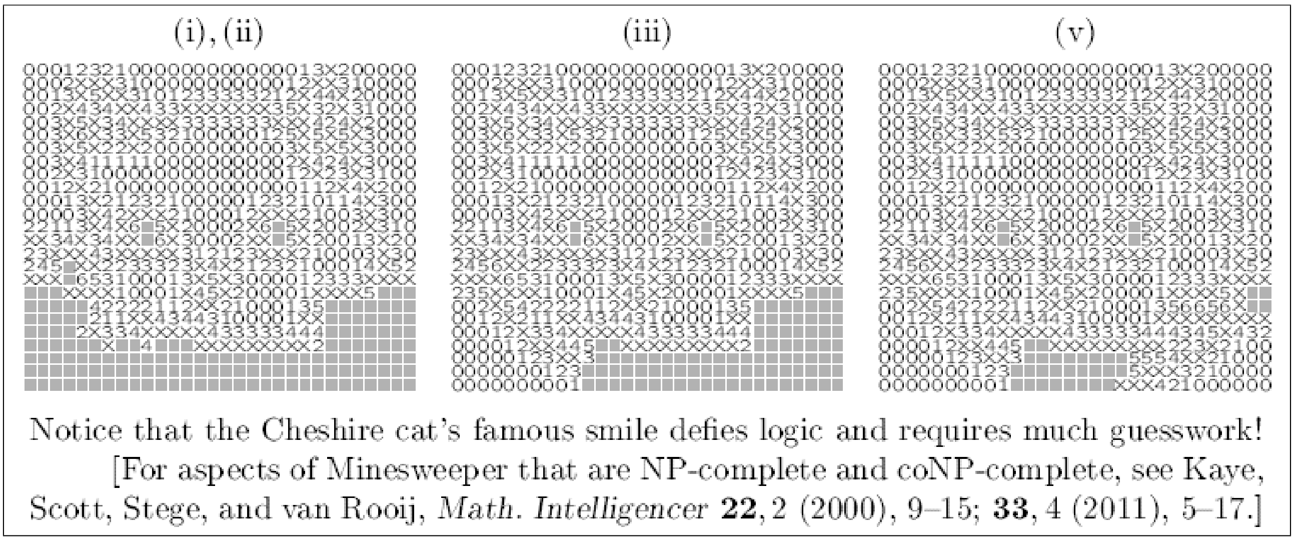
114. (a) From $x_{7,23}+x_{7,24}=x_{7,23}+x_{7,24}+x_{7,25}=x_{7,24}+x_{7,25}=1$ we deduce $x_{7,23}=x_{7,25}=0$ and $x_{7,24}=1$, revealing $n_{7,23}=n_{7,25}=5$. Now $x_{6,23}+x_{6,24}=x_{6,24}+x_{6,25}=x_{4,24}+x_{5,24}+x_{6,24}+x_{6,25}=1$; hence $x_{4,24}=x_{5,24}=0$, revealing $n_{4,24}=n_{5,24}=2$. So $x_{6,23}=x_{6,25}=0$, and the rest is easy.

(b) Let $y_{i,j}$ mean "cell (i,j) has been probed safely, revealing $n_{i,j}$." Consider the clauses C obtained by appending $\bar{y}_{i,j}$ to each clause of the symmetric function $[\sum_{i'=i-1}^{i+1}\sum_{j'=j-1}^{j+1}x_{i',j'}=n_{i,j}]$, for all i,j with $x_{i,j}=0$. Also include $(\bar{x}_{i,j}\vee\bar{y}_{i,j})$, as well as clauses for the symmetric function $S_N(x)$ if we're told the total number N of mines.

Given any subset F of mine-free cells, the clauses $C_F=C\wedge\bigwedge\{y_{i,j}\mid(i,j)\in F\}$ are satisfiable precisely by the configurations of mines that are consistent with the data $\{n_{i,j}\mid(i,j)\in F\}$. Therefore cell (i,j) is safe if and only if $C_F\wedge x_{i,j}$ is unsatisfiable.

A simple modification of Algorithm C can be used to "grow" F until no further safe cells can be added: Given a solution to C_F for which neither $x_{i,j}$ nor $\bar{x}_{i,j}$ was obtained at root level (level 0), we can try to find a "flipped" solution by using the complemented value as the decision at level 1. Such a solution will be found if and only if the flipped value is consistent; otherwise the unflipped value will have been forced at level 0. By changing default polarities we can favor solutions that flip many variables at once. Whenever a literal $\bar{x}_{i,j}$ is newly deduced at root level, we can force $y_{i,j}$ to be true, thus adding (i,j) to F. We reach an impasse when a set of solutions has been obtained for C_F that covers both settings of every unforced $x_{i,j}$.

For problem (i) we start with $F=\{(1,1)\}$, etc. Case (iv) by itself uncovers only 56 cells in the lower right corner. The other results, each obtained in <6 Gμ, are:



(i), (ii)

```
0001232100000000000013X2000000
0002XXX3100000000000012XX31000
0013X5XX3101233333212X44X20000
002X434XX433XXXXXXX35X32X31000
003X5X34XXXX3333333XXX424X2000
003X6X33X532100000125X5X5X3000
003X5X22X20000000000003X5X5X3000
003X41111100000000002X424X3000
002X3100000000000000012X23X3100
0012X2100000000000000112X4X200
00013X21232100000123210114X300
00003X42XXX2100012XXX21003X300
22113X4X6■5X20002X6■5X2002X310
XX34X34XX■6X30002XX■5X20013X20
23XXX43XXXXX312123XXX210003X30
245■XX2233323X4X21232100014X52
XXX■653100013X5X300001233 3XXXX
■■■XXXX10001X45X200001XXXX5■■■
■■■■■42222112XX21000135■■■■■■■
■■■■■211XX43443100001XX■■■■■■■
■■■■2X334XXXXX433333444■■■■■■■
■■■■■■X■■4■■■XXXXXXXXX2■■■■■■■
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
```

(iii)

```
0001232100000000000013X2000000
0002XXX3100000000000012XX31000
0013X5XX3101233333212X44X20000
002X434XX433XXXXXXX35X32X31000
003X5X34XXXX3333333XXX424X2000
003X6X33X532100000125X5X5X3000
003X5X22X20000000000003X5X5X3000
003X41111100000000002X424X3000
002X3100000000000000012X23X3100
0012X2100000000000000112X4X200
00013X21232100000123210114X300
00003X42XXX2100012XXX21003X300
22113X4X6■5X20002X6■5X2002X310
XX34X34XX■6X30002XX■5X20013X20
23XXX43XXXXX312123XXX210003X30
2456XX2233323X4X21232100014X52
XXXX653100013X5X30000123 33XXXX
235XXXX10001X45X200001XXXX5■■■
002X542222112XX21000135■■■■■■■
0012X211XX43443100001XX■■■■■■■
00012X334XXXXX433333444■■■■■■■
000012XX445■■XXXXXXXXX2■■■■■■■
00000123XX3■■■■■■■■■■■■■■■■■■■
0000000123■■■■■■■■■■■■■■■■■■■■
0000000001■■■■■■■■■■■■■■■■■■■■
```

(v)

```
0001232100000000000013X2000000
0002XXX3100000000000012XX31000
0013X5XX3101233333212X44X20000
002X434XX433XXXXXXX35X32X31000
003X5X34XXXX3333333XXX424X2000
003X6X33X532100000125X5X5X3000
003X5X22X20000000000003X5X5X3000
003X41111100000000002X424X3000
002X3100000000000000012X23X3100
0012X2100000000000000112X4X200
00013X21232100000123210114X300
00003X42XXX2100012XXX21003X300
22113X4X6■5X20002X6■5X2002X310
XX34X34XX■6X30002XX■5X20013X20
23XXX43XXXXX312123XXX210003X30
2456XX2233323X4X21232100014X52
XXXX653100013X5X30000123 33XXXX
235XXXX10001X45X200001XXXX5X■■
002X542222112XX210001356656X■■
0012X211XX43443100001XXXXXXXXX
00012X334XXXXX433333444345X432
000012XX445■■XXXXXXXXX223X2100
00000123XX3■■■■■■■■5554XX21000
0000000123■■■■■■■■■5XXX3210000
0000000001■■■■■■■■XXX421000000
```

Notice that the Cheshire cat's famous smile defies logic and requires much guesswork!

[For aspects of Minesweeper that are NP-complete and coNP-complete, see Kaye, Scott, Stege, and van Rooij, *Math. Intelligencer* **22**, 2 (2000), 9–15; **33**, 4 (2011), 5–17.]

3.10.5 Cracking Minesweeper with SAT solver and sorting network

The SAT version of this program used Mathematica-generated POPCNT functions: 3.10.2.

Now what if I locked on a desert island again, with no Internet and Wolfram Mathematica?

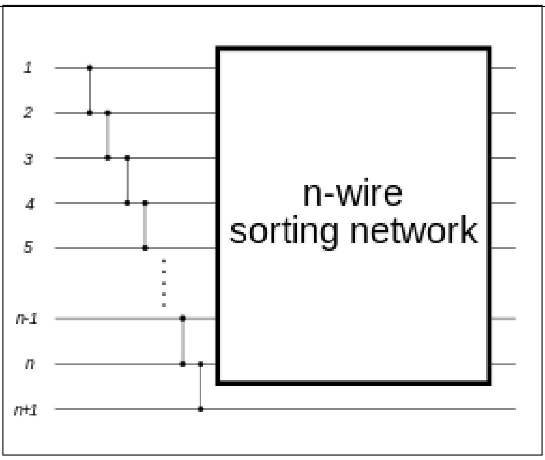
Here is another way of solving it using SAT solver. The main problem is to count bits around a cell.

Here (4.8) I described sorting networks shortly.

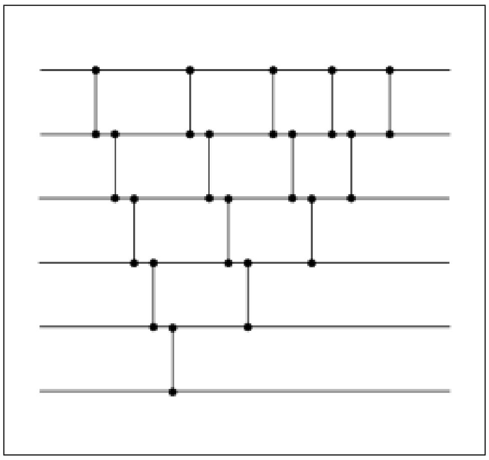
They can be used for sorting boolean values. 01101 will become 00111, 10001 -> 00011, etc. We will count bits using sorting network.

My implementation is a simplest *bubble sort* and not the one the most optimized I described earlier. It's created recursively, as shown in Wikipedia[6].

[6] https://en.wikipedia.org/wiki/Sorting_network



The resulting 6-wire network is:



Now the comparator/swapper. How do we compare/swap two boolean variables, A and B?

```
A B out1 out2
0 0    0    0
0 1    0    1
1 0    0    1
1 1    1    1
```

As you can deduce effortlessly, out1 is just an AND, out2 is OR.

And here is all functions creating sorting network in my SAT_lib Python library:

```
    def sort_unit(self, a, b):
        return self.OR_list([a,b]), self.AND(a,b)

    def sorting_network(self, lst:List[int]) -> List[int]:
        lst=copy.deepcopy(lst)
        lst_len=len(lst)
        for i in range(lst_len):
            for j in range((lst_len-i)-1):
                x, y = self.sort_unit(lst[j], lst[j+1])
                lst[j], lst[j+1] = x, y
        return lst
```

(https://smt.st/current_tree/libs/SAT_lib.py)

Now if you will look closely on the output of sorting network, it looks like a thermometer, isn't it? This is indeed *unary coding*, or *thermometer code*, where 1 is encoded as 1, 2 as 11... 4 as 1111, etc. Who need such a wasteful code? For 9 inputs/outputs, we can afford it so far.

In other words, sorting network is a device counting input bits and giving output in unary coding.

Also, we don't need to add 9 constraints for each variable. Only two will suffice, one False and one True, because we are only interesting in the *level* of a thermometer.

```
    def POPCNT(self, n:int, vars:List[str]):
        sorted=self.sorting_network(vars)
        self.fix_always_false(sorted[n])
        if n!=0:
            self.fix_always_true(sorted[n-1])
```

(https://smt.st/current_tree/libs/SAT_lib.py)

And the whole source code:

```
#!/usr/bin/python3

import SAT_lib
from typing import List

WIDTH=9
HEIGHT=9

known=[
"01?10001?",
"01?100011",
"011100000",
"000000000",
"111110011",
"????1001?",
"????3101?",
"?????211?",
"?????????"]

def chk_bomb(row:int, col:int):
    s=SAT_lib.SAT_lib()
    vars=[[s.create_var() for c in range(WIDTH+2)] for r in range(HEIGHT+2)]

    # make empty border
    # all variables are negated (because they must be False)
    for c in range(WIDTH+2):
        s.fix_always_false(vars[0][c])
        s.fix_always_false(vars[HEIGHT+1][c])
    for r in range(HEIGHT+2):
        s.fix_always_false(vars[r][0])
        s.fix_always_false(vars[r][WIDTH+1])

    for r in range(1,HEIGHT+1):
        for c in range(1,WIDTH+1):
            t=known[r-1][c-1]
            if t in "012345678":
                # cell at r, c is empty (False):
                s.fix_always_false(vars[r][c])
                # we need an empty border so the following expression would work for
                   all possible cells:
                neighbours=[vars[r-1][c-1], vars[r-1][c], vars[r-1][c+1], vars[r][c
                   -1],
                        vars[r][c+1], vars[r+1][c-1], vars[r+1][c], vars[r+1][c+1]]
                s.POPCNT(int(t), neighbours)

    # place a bomb
    s.fix_always_true (vars[row][col])

    if s.solve()==False:
        print ("row=%d, col=%d, unsat!" % (row, col))

for r in range(1,HEIGHT+1):
    for c in range(1,WIDTH+1):
        if known[r-1][c-1]=="?":
            chk_bomb(r, c)
```

As before, this is a list of Minesweeper cells you can safely click on:

```
row=1, col=3, unsat!
row=6, col=2, unsat!
row=6, col=3, unsat!
row=7, col=4, unsat!
row=7, col=9, unsat!
row=8, col=9, unsat!
```

However, it performs several times slower than the version with Mathematica-generated POPCNT functions, which is the fastest version so far...

Nevertheless, sorting networks has important place in SAT/SMT world. By fixing a "level" of a thermometer using a single constraint, it's possible to add PB (pseudo-boolean) constraints, like, "x>=10" (you need just to force a "level" to be always higher or equal than 10).

3.10.6 Cracking Minesweeper: by bruteforce

Now here is a bruteforce solver I wrote for fun: https://smt.st/current_tree/equations/minesweeper/6_brute/MS.py.

I doesn't use any external library or solver. However, it's painfully slow, it takes several minutes to find safe cells on 9*9 field. Still, it can serve as a demonstration.

This is where SAT/SMT solvers excels: they can find faster ways than bruteforce...

3.10.7 More theory

In Numb3rs TV series, Dr.Charlie Eppes tried to solve "Minesweeper game": https://en.wikipedia.org/wiki/Uncertainty_Principle_(Numbers), http://pi.math.cornell.edu/~numb3rs/luthy/num104.html.

What he is tried to do, is to find faster/better way to solve this game, by finding better methods to solve NP-problems in general.

Solving Minesweeper is NP-complete: http://simon.bailey.at/random/kaye.minesweeper.pdf, http://web.mat.bham.ac.uk/R.W.Kaye/minesw/ordmsw.htm, http://web.mat.bham.ac.uk/R.W.Kaye/minesw/ASE2003.pdf, http://math.oregonstate.edu/~math_reu/proceedings/REU_Proceedings/Proceedings2003/2003MK.pdf.

Our way to solve it is also called "Minesweeper Consistency Problem" [7]. We did it using system of integer equation. And solving system of integer equation is NP-complete as well.

Find better way to solve Minesweeper, and you probably will be able to solve other NP-problems!

3.10.8 Further reading

Proofsweeper – Play Minesweeper by formally proving your moves in Idris: https://github.com/A1kmm/proofsweeper.

Raphaël Collet – PlayingtheMinesweeperwithConstraints: https://www.info.ucl.ac.be/~pvr/minesweeper.pdf.

3.11 LCG

3.11.1 Cracking LCG with Z3

There are well-known weaknesses of LCG [8], but let's see, if it would be possible to crack it straightforwardly, without any special knowledge. We will define all relations between LCG states in terms of Z3. Here is a test program:

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

int main()
{
```

[7] https://www.claymath.org/sites/default/files/minesweeper.pdf

[8] http://en.wikipedia.org/wiki/Linear_congruential_generator#Advantages_and_disadvantages_of_LCGs, http://www.reteam.org/papers/e59.pdf, http://stackoverflow.com/questions/8569113/why-1103515245-is-used-in-rand/8574774#8574774

```
        int i;

        srand(time(NULL));

        for (i=0; i<10; i++)
                printf ("%d\n", rand()%100);
};
```

It is printing 10 pseudorandom numbers in 0..99 range:

```
37
29
74
95
98
40
23
58
61
17
```

Let's say we are observing only 8 of these numbers (from 29 to 61) and we need to predict next one (17) and/or previous one (37).

The program is compiled using MSVC 2013 (I choose it because its LCG is simpler than that in Glib):

```
.text:0040112E rand              proc near
.text:0040112E                   call    __getptd
.text:00401133                   imul    ecx, [eax+0x14], 214013
.text:0040113A                   add     ecx, 2531011
.text:00401140                   mov     [eax+14h], ecx
.text:00401143                   shr     ecx, 16
.text:00401146                   and     ecx, 7FFFh
.text:0040114C                   mov     eax, ecx
.text:0040114E                   retn
.text:0040114E rand              endp
```

Let's define LCG in Z3Py:

```
#!/usr/bin/python
from z3 import *

output_prev = BitVec('output_prev', 32)
state1 = BitVec('state1', 32)
state2 = BitVec('state2', 32)
state3 = BitVec('state3', 32)
state4 = BitVec('state4', 32)
state5 = BitVec('state5', 32)
state6 = BitVec('state6', 32)
state7 = BitVec('state7', 32)
state8 = BitVec('state8', 32)
state9 = BitVec('state9', 32)
state10 = BitVec('state10', 32)
output_next = BitVec('output_next', 32)

s = Solver()

s.add(state2 == state1*214013+2531011)
s.add(state3 == state2*214013+2531011)
s.add(state4 == state3*214013+2531011)
s.add(state5 == state4*214013+2531011)
s.add(state6 == state5*214013+2531011)
s.add(state7 == state6*214013+2531011)
s.add(state8 == state7*214013+2531011)
```

```
s.add(state9 == state8*214013+2531011)
s.add(state10 == state9*214013+2531011)

s.add(output_prev==URem((state1>>16)&0x7FFF,100))
s.add(URem((state2>>16)&0x7FFF,100)==29)
s.add(URem((state3>>16)&0x7FFF,100)==74)
s.add(URem((state4>>16)&0x7FFF,100)==95)
s.add(URem((state5>>16)&0x7FFF,100)==98)
s.add(URem((state6>>16)&0x7FFF,100)==40)
s.add(URem((state7>>16)&0x7FFF,100)==23)
s.add(URem((state8>>16)&0x7FFF,100)==58)
s.add(URem((state9>>16)&0x7FFF,100)==61)
s.add(output_next==URem((state10>>16)&0x7FFF,100))

print(s.check())
print(s.model())
```

URem states for *unsigned remainder*. It works for some time and gave us correct result!

```
sat
[state3 = 2276903645,
 state4 = 1467740716,
 state5 = 3163191359,
 state7 = 4108542129,
 state8 = 2839445680,
 state2 = 998088354,
 state6 = 4214551046,
 state1 = 1791599627,
 state9 = 548002995,
 output_next = 17,
 output_prev = 37,
 state10 = 1390515370]
```

I added ≈ 10 states to be sure result will be correct. It may be not in case of smaller set of information.

That is the reason why LCG is not suitable for any security-related task. This is why cryptographically secure pseudorandom number generators exist: they are designed to be protected against such simple attack. Well, at least if NSA[9] don't get involved [10].

Security tokens like "RSA SecurID" can be viewed just as CPRNG[11] with a secret seed. It shows new pseudorandom number each minute, and the server can predict it, because it knows the seed. Imagine if such token would implement LCG—it would be much easier to break!

3.11.2 Can rand() generate 10 consecutive zeroes?

I've always been wondering, if it's possible or not. As of simplest linear congruential generator from MSVC's rand(), I could get a state at which rand() will output 8 zeroes modulo 10:

```
#!/usr/bin/python3

from z3 import *

state1 = BitVec('state1', 32)
state2 = BitVec('state2', 32)
state3 = BitVec('state3', 32)
state4 = BitVec('state4', 32)
state5 = BitVec('state5', 32)
state6 = BitVec('state6', 32)
state7 = BitVec('state7', 32)
state8 = BitVec('state8', 32)
state9 = BitVec('state9', 32)
```

[9] National Security Agency

[10] https://en.wikipedia.org/wiki/Dual_EC_DRBG

[11] Cryptographically Secure Pseudorandom Number Generator

```
s = Solver()

s.add(state2 == state1*214013+2531011)
s.add(state3 == state2*214013+2531011)
s.add(state4 == state3*214013+2531011)
s.add(state5 == state4*214013+2531011)
s.add(state6 == state5*214013+2531011)
s.add(state7 == state6*214013+2531011)
s.add(state8 == state7*214013+2531011)
s.add(state9 == state8*214013+2531011)

s.add(URem((state2>>16)&0x7FFF,10)==0)
s.add(URem((state3>>16)&0x7FFF,10)==0)
s.add(URem((state4>>16)&0x7FFF,10)==0)
s.add(URem((state5>>16)&0x7FFF,10)==0)
s.add(URem((state6>>16)&0x7FFF,10)==0)
s.add(URem((state7>>16)&0x7FFF,10)==0)
s.add(URem((state8>>16)&0x7FFF,10)==0)
s.add(URem((state9>>16)&0x7FFF,10)==0)

print(s.check())
print(s.model())
```

```
sat
[state3 = 1181667981,
 state4 = 342792988,
 state5 = 4116856175,
 state7 = 1741999969,
 state8 = 3185636512,
 state2 = 1478548498,
 state6 = 4036911734,
 state1 = 286227003,
 state9 = 1700675811]
```

This is a case if, in some video game, you'll find a code:

```
for (int i=0; i<8; i++)
    printf ("%d\n", rand() % 10);
```

... and at some point, this piece of code can generate 8 zeroes in row, if the state will be 286227003 (decimal).

Just checked this piece of code in MSVC 2015:

```
// MSVC 2015 x86

#include <stdio.h>

int main()
{
        srand(286227003);

        for (int i=0; i<8; i++)
                printf ("%d\n", rand() % 10);
};
```

Yes, its output is 8 zeroes!

What about other modulos?

I can get 4 consecutive zeroes modulo 100:

```
#!/usr/bin/python3

from z3 import *

```

```
state1 = BitVec('state1', 32)
state2 = BitVec('state2', 32)
state3 = BitVec('state3', 32)
state4 = BitVec('state4', 32)
state5 = BitVec('state5', 32)

s = Solver()

s.add(state2 == state1*214013+2531011)
s.add(state3 == state2*214013+2531011)
s.add(state4 == state3*214013+2531011)
s.add(state5 == state4*214013+2531011)

s.add(URem((state2>>16)&0x7FFF,100)==0)
s.add(URem((state3>>16)&0x7FFF,100)==0)
s.add(URem((state4>>16)&0x7FFF,100)==0)
s.add(URem((state5>>16)&0x7FFF,100)==0)

print(s.check())
print(s.model())
```

```
sat
[state3 = 635704497,
 state4 = 1644979376,
 state2 = 1055176198,
 state1 = 3865742399,
 state5 = 1389375667]
```

However, 4 consecutive zeroes modulo 100 is impossible (given these constants at least), this gives "unsat": https://smt.st/current_tree/equations/LCG/LCG100_v1.py.

... and 3 consecutive zeroes modulo 1000:

```
#!/usr/bin/python3

from z3 import *

state1 = BitVec('state1', 32)
state2 = BitVec('state2', 32)
state3 = BitVec('state3', 32)
state4 = BitVec('state4', 32)

s = Solver()

s.add(state2 == state1*214013+2531011)
s.add(state3 == state2*214013+2531011)
s.add(state4 == state3*214013+2531011)

s.add(URem((state2>>16)&0x7FFF,1000)==0)
s.add(URem((state3>>16)&0x7FFF,1000)==0)
s.add(URem((state4>>16)&0x7FFF,1000)==0)

print(s.check())
print(s.model())
```

```
sat
[state3 = 1179663182,
 state2 = 720934183,
 state1 = 4090229556,
 state4 = 786474201]
```

What if we could use rand()'s output without division? Which is in 0..0x7fff range (i.e., 15 bits)? As it can be checked quickly, 2 zeroes at output is possible:

```
#!/usr/bin/env python3

from z3 import *

state1 = BitVec('state1', 32)
state2 = BitVec('state2', 32)
state3 = BitVec('state3', 32)

s = Solver()

s.add(state2 == state1*214013+2531011)
s.add(state3 == state2*214013+2531011)

s.add((state2>>16)&0x7FFF==0)
s.add((state3>>16)&0x7FFF==0)

print(s.check())
print(s.model())
```

```
sat
[state2 = 20057, state1 = 3385131726, state3 = 22456]
```

UNIX time and srand(time(NULL))

Given the fact that it's highly popular to initialize LCG PRNG with UNIX time (i.e., `srand(time(NULL))`), you can probably calculate a moment in time so that LCG PRNG will be initialized as you want to.

For example, can we get a moment in time from now (5-Dec-2017) till 12-Dec-2017 (that is one week from now), when, if initialized by UNIX time, rand() will output as many similar numbers (modulo 10), as possible?

```
#!/usr/bin/python3

from z3 import *

state1 = BitVec('state1', 32)
state2 = BitVec('state2', 32)
state3 = BitVec('state3', 32)
state4 = BitVec('state4', 32)
state5 = BitVec('state5', 32)
state6 = BitVec('state6', 32)
state7 = BitVec('state7', 32)

s = Solver()

s.add(state1>=1512499124) # Tue Dec  5 20:38:44 EET 2017
s.add(state1<=1513036800) # Tue Dec 12 02:00:00 EET 2017

s.add(state2 == state1*214013+2531011)
s.add(state3 == state2*214013+2531011)
s.add(state4 == state3*214013+2531011)
s.add(state5 == state4*214013+2531011)
s.add(state6 == state5*214013+2531011)
s.add(state7 == state6*214013+2531011)

c = BitVec('c', 32)

s.add(URem((state2>>16)&0x7FFF,10)==c)
s.add(URem((state3>>16)&0x7FFF,10)==c)
s.add(URem((state4>>16)&0x7FFF,10)==c)
s.add(URem((state5>>16)&0x7FFF,10)==c)
s.add(URem((state6>>16)&0x7FFF,10)==c)
```

```
s.add(URem((state7>>16)&0x7FFF,10)==c)

print(s.check())
print(s.model())
```

Yes:

```
sat
[state3 = 2234253076,
 state4 = 497021319,
 state5 = 4160988718,
 c = 3,
 state2 = 333151205,
 state6 = 46785593,
 state1 = 1512500810,
 state7 = 1158878744]
```

If `srand(time(NULL))` will be executed at `Tue Dec 5 21:06:50 EET 2017` (this precise second, UNIX time=1512500810), a next 6 `rand() % 10` lines will output six numbers of 3 in a row. Don't know if it useful or not, but you've got the idea.

etc:

The files: https://smt.st/current_tree/equations/LCG.

Further work: check glibc's `rand()`, Mersenne Twister, etc. A simple 32-bit LCG (as described) can be checked using simple brute-force, I think.

Fun story

The software checked protection key (dongle) randomly, from time to time. This code snippet is from a real one:

```
void init_all()
{
        ...

        srand(time(NULL));

        ...
};

...

void check_protection_thread()
{
        // get in 0..9 range
        int t=(int)((double)rand()/3276);
        if (t== 5)
        {
                check protection
        }
};
```

Perhaps, we can find the most optimal UNIX time to start the software, so the protection will not be checked as long as possible...

Further reading

Breaking JavaScript's PRNG (XorShift128+):
https://blog.securityevaluators.com/hacking-the-javascript-lottery-80cc437e3b7f.

3.11.3 Can rand() generate 100 consecutive zeroes? (SMT-LIB versions)

Let's implement this in terms of SMT-LIB language.

Version 1: unrolled

```
(set-logic QF_BV)
(set-info :smt-lib-version 2.0)

;(declare-const const1 (_ BitVec 32)) ; Z3's syntax
(declare-fun const1 () (_ BitVec 32))
(assert (= const1 (_ bv214013 32)))
;(declare-const const2 (_ BitVec 32)) ; Z3's syntax
(declare-fun const2 () (_ BitVec 32))
(assert (= const2 (_ bv2531011 32)))

(declare-fun state1 () (_ BitVec 32))
(declare-fun state2 () (_ BitVec 32))
(declare-fun state3 () (_ BitVec 32))
(declare-fun state4 () (_ BitVec 32))

(assert (= (bvadd (bvmul state1 const1) const2) state2))
(assert (= (bvadd (bvmul state2 const1) const2) state3)) ; isn't it redundant?
(assert (= (bvadd (bvmul state3 const1) const2) state4)) ; isn't it redundant?

(assert (= (bvurem (bvand (bvlshr state2 (_ bv16 32)) #x00007fff) (_ bv1000 32)) (_
   bv0 32)))
(assert (= (bvurem (bvand (bvlshr state3 (_ bv16 32)) #x00007fff) (_ bv1000 32)) (_
   bv0 32))) ; isn't it redundant?
(assert (= (bvurem (bvand (bvlshr state4 (_ bv16 32)) #x00007fff) (_ bv1000 32)) (_
   bv0 32))) ; isn't it redundant?

; two correct solutions:
;(assert (not (= state1 #xf3cbf334)))
;(assert (not (= state1 #x73cbf334)))

(check-sat)
(get-model)
```

QF_BV logic means ”quantifier free, bitvector”.

So far so good, but a bit redundant. Can we wrap these redundant parts into function(s)?

Version 2: functions

We can define functions in SMT-LIB language:

```
(set-logic QF_BV)
(set-info :smt-lib-version 2.0)

(declare-fun const1 () (_ BitVec 32))
(assert (= const1 (_ bv214013 32)))
(declare-fun const2 () (_ BitVec 32))
(assert (= const2 (_ bv2531011 32)))

(declare-fun state1 () (_ BitVec 32))
(declare-fun state2 () (_ BitVec 32))
(declare-fun state3 () (_ BitVec 32))
(declare-fun state4 () (_ BitVec 32))

(define-fun PRNG_step
        ((state (_ BitVec 32))) ; list of arguments
        (_ BitVec 32)           ; type of return value
        (bvadd (bvmul state const1) const2) ; body
)

(define-fun PRNG_get_result_modulo
        ((state (_ BitVec 32))) ; list of arguments
        (_ BitVec 32)           ; type of return value
        (bvand (bvlshr state (_ bv16 32)) #x00007fff) ; body
)

(define-fun PRNG_get_result_modulo_1000
        ((state (_ BitVec 32))) ; list of arguments
        (_ BitVec 32)           ; type of return value
        (bvurem (PRNG_get_result_modulo state) (_ bv1000 32)) ; body
)

(assert (= (PRNG_step state1) state2))
(assert (= (PRNG_step state2) state3))
(assert (= (PRNG_step state3) state4))

(assert (= (PRNG_get_result_modulo_1000 state2) (_ bv0 32)))
(assert (= (PRNG_get_result_modulo_1000 state3) (_ bv0 32)))
(assert (= (PRNG_get_result_modulo_1000 state4) (_ bv0 32)))

; two correct solutions:
;(assert (not (= state1 #xf3cbf334)))
;(assert (not (= state1 #x73cbf334)))

(check-sat)
(get-model)
```

Version 3: function + array

Now arrays. You see, declaring many ”state” variables is also redundant. Can’t we use arrays here?

```
(set-logic QF_ABV)
(set-info :smt-lib-version 2.0)

(declare-fun const1 () (_ BitVec 32))
(assert (= const1 (_ bv214013 32)))
(declare-fun const2 () (_ BitVec 32))
(assert (= const2 (_ bv2531011 32)))

(declare-const states (Array (_ BitVec 4) (_ BitVec 32)))
(declare-fun init () (_ BitVec 32))
(assert (= init (select states #x1)))
(assert (not (= init (_ bv1942745908 32))))

(define-fun PRNG_step
        ((state (_ BitVec 32))) ; list of arguments
        (_ BitVec 32)           ; type of return value
        (bvadd (bvmul state const1) const2) ; body
)

(define-fun PRNG_get_result_modulo
        ((state (_ BitVec 32))) ; list of arguments
        (_ BitVec 32)           ; type of return value
        (bvand (bvlshr state (_ bv16 32)) #x00007fff) ; body
)

(define-fun PRNG_get_result_modulo_1000
        ((state (_ BitVec 32))) ; list of arguments
        (_ BitVec 32)           ; type of return value
        (bvurem (PRNG_get_result_modulo state) (_ bv1000 32)) ; body
)
```

```
(assert (= (PRNG_step (select states #x1)) (select states #x2)))
(assert (= (PRNG_step (select states #x2)) (select states #x3)))
(assert (= (PRNG_step (select states #x3)) (select states #x4)))

(assert (= (PRNG_get_result_modulo_1000 (select states #x2)) (_ bv0 32)))
(assert (= (PRNG_get_result_modulo_1000 (select states #x3)) (_ bv0 32)))
(assert (= (PRNG_get_result_modulo_1000 (select states #x4)) (_ bv0 32)))

(check-sat)
(get-model)
```

Here we use the `QF_ABV` SMT logic: same as in `QF_BV`, but `A` means *arrays*.
We use arrays also in this example: 5.2.2.
Here we define relationships between various elements of array:

```
(assert (= (PRNG_step (select states #x1)) (select states #x2)))
(assert (= (PRNG_step (select states #x2)) (select states #x3)))
(assert (= (PRNG_step (select states #x3)) (select states #x4)))
```

Here we *read* from array:

```
(assert (= (PRNG_get_result_modulo_1000 (select states #x2)) (_ bv0 32)))
(assert (= (PRNG_get_result_modulo_1000 (select states #x3)) (_ bv0 32)))
(assert (= (PRNG_get_result_modulo_1000 (select states #x4)) (_ bv0 32)))
```

Version 4: UF[12] + array

We can also define function using UF. These functions that has no body. This is a bit unusual concept for a programmer accustomed to languages derived from Algol or Lisp.

However, you can describe behaviour of UF using additional constraints.

```
;(set-logic AUFBV) ; Boolector can't parse this
(set-info :smt-lib-version 2.0)

(declare-fun const1 () (_ BitVec 32))
(assert (= const1 (_ bv214013 32)))
(declare-fun const2 () (_ BitVec 32))
(assert (= const2 (_ bv2531011 32)))

(declare-fun rand ((_ BitVec 32)) (_ BitVec 32))
(assert (forall ((x (_ BitVec 32)))
        (= (rand x) (bvadd (bvmul x const1) const2))))

(declare-const states (Array (_ BitVec 4) (_ BitVec 32)))

(declare-fun init () (_ BitVec 32))
(assert (= init (select states #x1)))

(assert (= (select states #x2) (rand (select states #x1))))
(assert (= (select states #x3) (rand (select states #x2))))
(assert (= (select states #x4) (rand (select states #x3))))

(declare-fun get_result ((_ BitVec 32)) (_ BitVec 32))
(assert (forall ((x (_ BitVec 32)))
        (= (get_result x) (bvurem (bvand (bvlshr x (_ bv16 32)) #x00007fff) (_ bv1000
            32)))))

(assert (= (get_result (select states #x2)) (_ bv0 32)))
(assert (= (get_result (select states #x3)) (_ bv0 32)))
(assert (= (get_result (select states #x4)) (_ bv0 32)))
```

[12] Uninterpreted Function

```
(check-sat)
(get-model)
```

Take a closer look:

```
(assert (forall ((x (_ BitVec 32)))
        (= (rand x) (bvadd (bvmul x const1) const2))))
```

In Plain English, this means: ”for all (32-bit) X’s, rand(x)’s value is always equals to (bvadd (bvmul X const1) const2))))”.

This is First-Order Logic. ”For all” is an *universal quantifier*, usually denoted as an ”A” letter upside down: ∀.

Since quantifiers exists in the example, another SMT logic is to be used: `AUFBV` (A stands for Arrays, UF for UF, BV stands for bitvector, but notice we dropped the `QF_` prefix).

UF’s are then *called*, as in Lisp language: ”(function_name arg1 arg2 ...)”.

Performance

(Time in seconds, running on my relic Intel Core 2 Duo T9400, clocked at 2.13GHz.)

SMT-solver	v1 (unrolled)	v2 (functions)	v3 (function + array)	v4 (UF + array)
STP 2.3.3	3.6	3.6	5.6	*
Boolector 3.2.1	8	8	5	3.5
CVC4 1.9-prerelease	21	21	16	5
Z3 4.8.9	23	23	26	133
Yices 2.6.2	6	6	11	*
MK85	209	*	*	*

*) Couldn’t manage to run it

Also, since the first version of the example is that simple, my toy-level bitblaster (MK85, that is based on picosat SAT solver) can solve it, slower by factor of 10x then others, but still...

Bruteforce

But what about bruteforce? Yes, you can solve this problem using simple bruteforce. 32-bit search space is small enough. But... this is what I can do in pure C:

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
        for (unsigned int seed=0; ; seed++)
        {
                unsigned int t=seed;
                t=t*214013 + 2531011;
                unsigned int a1=((t>>16)&0x7fff) % 1000;
                if (a1!=0)
                        continue;
                t=t*214013 + 2531011;
                unsigned int a2=((t>>16)&0x7fff) % 1000;
                if (a2!=0)
                        continue;
                t=t*214013 + 2531011;
                unsigned int a3=((t>>16)&0x7fff) % 1000;
                if (a3==0)
                {
                        printf ("seed: 0x%x\n", seed);
                        exit(0);
                };
                if (seed==0xffffffff)
                        break;
        };
};
```

If compiled with `GCC -O3`, it will run for 12 seconds on the same CPU. Faster than Z3 and CVC4, but slower than STP and Boolector! Modulo division is a heavy operation, but it seems, these SMT solvers can find ways to cut the search space!

The files

https://smt.st/current_tree/equations/LCG

3.11.4 Other PRNGs

In July 2022 I tried to solve Project Euler number 803[13], but found that SMT solvers are incapable to deal with 48-bit LCG.

Bruteforce was much more faster.

I probably should remove this section, but didn't. I'll leave it to stress the fact that sometimes, bruteforce may be faster than SAT/SMT solvers. So keep this in mind.

3.12 Integer factorization using Z3 SMT solver

Integer factorization is method of breaking a composite (non-prime number) into prime factors. Like 12345 = 3*4*823.

Though for small numbers, this task can be accomplished by Z3:

```
#!/usr/bin/env python3

import random, functools
from z3 import *
from operator import mul

def factor(n):
    print ("factoring",n)

    in1,in2,out=Ints('in1 in2 out')

    s=Solver()
    s.add(out==n)
    s.add(in1*in2==out)
    # inputs cannot be negative and must be non-1:
    s.add(in1>1)
    s.add(in2>1)

    if s.check()==unsat:
        print (n,"is prime (unsat)")
        return [n]
    if s.check()==unknown:
        print (n,"is probably prime (unknown)")
        return [n]

    m=s.model()
    # get inputs of multiplier:
    in1_n=m[in1].as_long()
    in2_n=m[in2].as_long()

    print ("factors of", n, "are", in1_n, "and", in2_n)
    # factor factors recursively:
    rt=sorted(factor (in1_n) + factor (in2_n))
    # self-test:
    assert functools.reduce(mul, rt, 1)==n
    return rt

# infinite test:
def test():
```

[13] https://projecteuler.net/problem=803

```
    while True:
        print (factor (random.randrange(1000000000)))

#test()

print (factor(1234567890))
#print (factor(12345))
```

(The source code: https://smt.st/current_tree/equations/factor_SMT/factor_z3.py)

When factoring 1234567890 recursively:

```
% time python z.py
factoring 1234567890
factors of 1234567890 are 342270 and 3607
factoring 342270
factors of 342270 are 2 and 171135
factoring 2
2 is prime (unsat)
factoring 171135
factors of 171135 are 3803 and 45
factoring 3803
3803 is prime (unsat)
factoring 45
factors of 45 are 3 and 15
factoring 3
3 is prime (unsat)
factoring 15
factors of 15 are 5 and 3
factoring 5
5 is prime (unsat)
factoring 3
3 is prime (unsat)
factoring 3607
3607 is prime (unsat)
[2, 3, 3, 5, 3607, 3803]
python z.py  19.30s user 0.02s system 99% cpu 19.443 total
```

So, 1234567890 = 2*3*3*5*3607*3803.

One important note: there is no primality test, no lookup tables, etc. Prime number is a number for which ”x*y=prime” (where x>1 and y>1) diophantine equation (which allows only integers in solution) has no solutions. It can be solved for real numbers, though.

Z3 is not yet good enough for non-linear integer arithmetic and sometimes returns ”unknown” instead of ”unsat”, but, as Leonardo de Moura (one of Z3’s author) commented about this:

```
...Z3 will solve the problem as a real problem. If no real solution is found, we know
    there is no integer solution.
If a solution is found, Z3 will check if the solution is really assigning integer
   values to integer variables.
If that is not the case, it will return unknown to indicate it failed to solve the
   problem.
```

(https://stackoverflow.com/questions/13898175/how-does-z3-handle-non-linear-integer-arithmetic)

Probably, this is the case: we getting ”unknown” in the case when a number cannot be factored, i.e., it’s prime.

It’s also very slow. Wolfram Mathematica can factor number around 2^{80} in a matter of seconds. Still, I’ve written this for demonstration.

The problem of breaking RSA is a problem of factorization of very large numbers, up to 2^{4096}. It’s currently not possible to do this in practice.

3.13 Integer factorization using SAT solver

In place of epigraph:

> It's somewhat mind-boggling to realize that numbers can be factored without using any number theory! No greatest common divisors, no applications of Fermat's theorems, etc., are anywhere in sight. We're providing no hints to the solver except for a bunch of Boolean formulas that operate almost blindly at the bit level. Yet factors are found.
>
> Of course we can't expect this method to compete with the sophisticated factorization algorithms of Section 4.5.4. But the problem of factoring does demonstrate the great versatility of clauses. And its clauses can be combined with other constraints that go well beyond any of the problems we've studied before.

(Donald Knuth, The Art of Computer Programming, section 7.2.2.2, page 10)

See also: integer factorization using Z3 SMT solver (3.12).

We are going to simulate electronic circuit of binary multiplier in SAT and then ask solver, what multiplier's inputs must be so the output will be a desired number? If this situation is impossible, the desired number is prime.

First we should build multiplier out of adders.

3.13.1 Binary adder in SAT

Simple binary adder usually consists of full-adders and one half-adder. These are basic elements of adders.

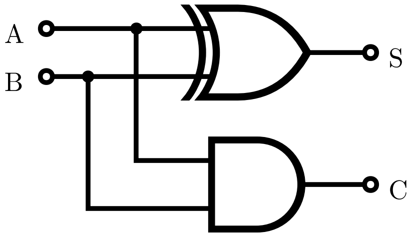


Figure 3.6: A half-adder. (The image has been taken from Wikipedia.)

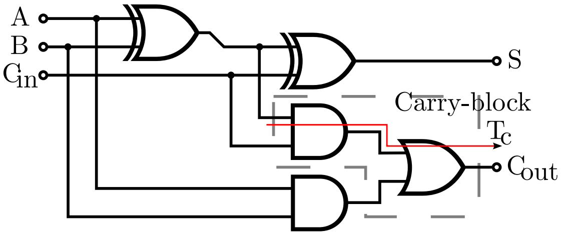


Figure 3.7: A full-adder. (The image has been taken from Wikipedia.)

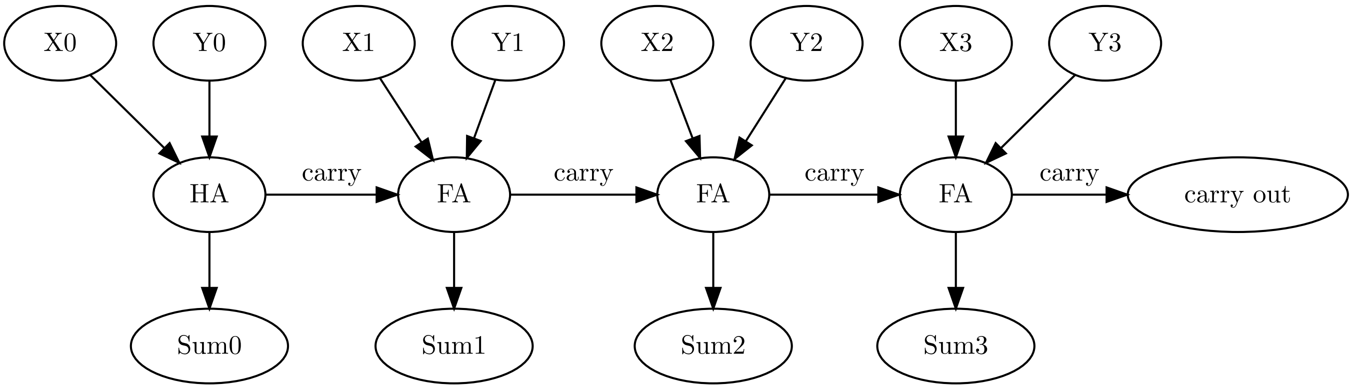


Figure 3.8: 4-bit ripple-carry adder

The ripple-carry adder can be used for most tasks.

A	B	Cin	Cout	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Table 3.1: The truth table for full-adder

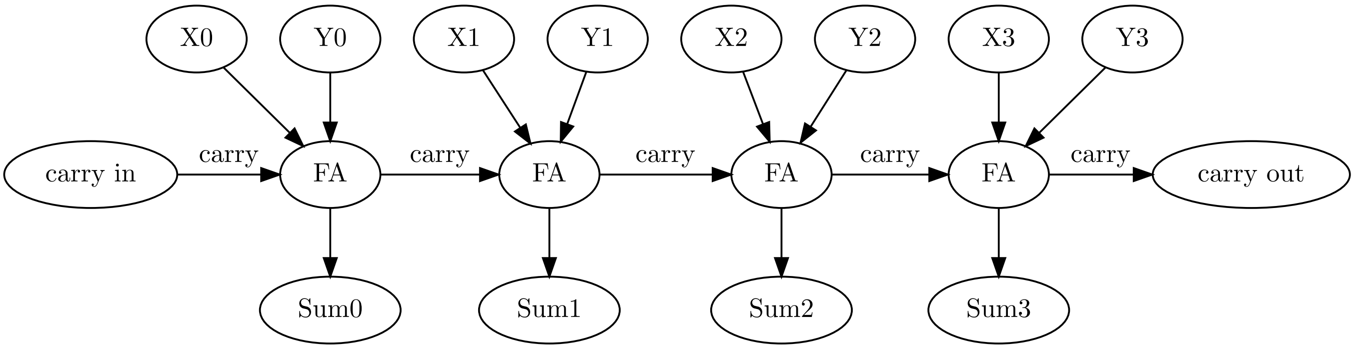


Figure 3.9: The ripple-carry 4-bit adder with carry-in

What carries are? 4-bit adder can sum up two numbers up to 0b1111 (15). 15+15=30 and this is 0b11110, i.e., 5 bits. Lowest 4 bits is a sum. 5th most significant bit is not a part of sum, but is a carry bit.

If you sum two numbers on x86 CPU, CF flag is a carry bit connected to ALU[14]. It is set if a resulting sum is bigger than it can be fit into result.

Now you can also need carry-in. Again, x86 CPU has ADC instruction, it takes CF flag state. It can be said, CF flag is connected to adder's carry-in input. Hence, combining two ADD and ADC instructions you can sum up 128 bits on 64-bit CPU.

By the way, this is a good explanation of "carry-ripple". The very first full-adder's result is depending on the carry-out of the previous full-adder. Hence, adders cannot work in parallel. This is a problem of simplest possible adder, other adders can solve this.

To represent full-adders in CNF form, we can use Wolfram Mathematica.

In Mathematica, I'm setting "->1" if row is correct and "->0" if not correct.

```
In[59]:= FaTbl = {{0, 0, 0, 0, 0} -> 1, {0, 0, 0, 0, 1} ->
   0, {0, 0, 0, 1, 0} -> 0, {0, 0, 0, 1, 1} -> 0, {0, 0, 1, 0, 0} ->
   0, {0, 0, 1, 0, 1} -> 1, {0, 0, 1, 1, 0} -> 0, {0, 0, 1, 1, 1} ->
   0, {0, 1, 0, 0, 0} -> 0, {0, 1, 0, 0, 1} -> 1, {0, 1, 0, 1, 0} ->
   0, {0, 1, 0, 1, 1} -> 0, {0, 1, 1, 0, 0} -> 0, {0, 1, 1, 0, 1} ->
   0, {0, 1, 1, 1, 0} -> 1, {0, 1, 1, 1, 1} -> 0, {1, 0, 0, 0, 0} ->
   0, {1, 0, 0, 0, 1} -> 1, {1, 0, 0, 1, 0} -> 0, {1, 0, 0, 1, 1} ->
   0, {1, 0, 1, 0, 0} -> 0, {1, 0, 1, 0, 1} -> 0, {1, 0, 1, 1, 0} ->
   1, {1, 0, 1, 1, 1} -> 0, {1, 1, 0, 0, 0} -> 0, {1, 1, 0, 0, 1} ->
   0, {1, 1, 0, 1, 0} -> 1, {1, 1, 0, 1, 1} -> 0, {1, 1, 1, 0, 0} ->
   0, {1, 1, 1, 0, 1} -> 0, {1, 1, 1, 1, 0} -> 0, {1, 1, 1, 1, 1} -> 1}

...

In[60]:= BooleanConvert[
 BooleanFunction[FaTbl, {a, b, cin, cout, s}], "CNF"]

Out[60]= (! a || ! b || ! cin || s) && (! a || ! b ||
   cout) && (! a || ! cin || cout) && (! a || cout || s) && (a || b ||
    cin || ! s) && (a || b || ! cout) && (a ||
```

[14] Arithmetic logic unit

```
   cin || ! cout) && (a || ! cout || ! s) && (! b || ! cin ||
   cout) && (! b || cout || s) && (b ||
   cin || ! cout) && (b || ! cout || ! s) && (! cin || cout ||
   s) && (cin || ! cout || ! s)
```

These clauses can be used as full-adder.

Here is it:

```
    # full-adder, as found by Mathematica using truth table:
    def FA (self, a,b,cin):
        s=self.create_var()
        cout=self.create_var()

        self.add_clause([self.neg(a), self.neg(b), self.neg(cin), s])
        self.add_clause([self.neg(a), self.neg(b), cout])
        self.add_clause([self.neg(a), self.neg(cin), cout])
        self.add_clause([self.neg(a), cout, s])
        self.add_clause([a, b, cin, self.neg(s)])
        self.add_clause([a, b, self.neg(cout)])
        self.add_clause([a, cin, self.neg(cout)])
        self.add_clause([a, self.neg(cout), self.neg(s)])
        self.add_clause([self.neg(b), self.neg(cin), cout])
        self.add_clause([self.neg(b), cout, s])
        self.add_clause([b, cin, self.neg(cout)])
        self.add_clause([b, self.neg(cout), self.neg(s)])
        self.add_clause([self.neg(cin), cout, s])
        self.add_clause([cin, self.neg(cout), self.neg(s)])

        return s, cout
```

And the adder:

```
    # bit order: [MSB..LSB]
    # n-bit adder:
    def adder(self, X,Y):
        assert len(X)==len(Y)
        # first full-adder could be half-adder
        # start with lowest bits:
        inputs=my_utils.rvr(list(zip(X,Y)))
        carry=self.const_false
        sums=[]
        for pair in inputs:
            # "carry" variable is replaced at each iteration.
            # so it is used in the each FA() call from the previous FA() call.
            s, carry = self.FA(pair[0], pair[1], carry)
            sums.append(s)
        return my_utils.rvr(sums), carry
```

3.13.2 Binary multiplier in SAT

Remember school-level long division? This multiplier works in a same way, but for binary digits.

Here is example of multiplying 0b1101 (X) by 0b0111 (Y):

```
         LSB
          |
          v
       1101 <- X
       -------
LSB 0|    0000
    1|   1101
    1|  1101
    1| 1101
    ^
    |
    Y
```

If bit from Y is zero, a row is zero. If bit from Y is non-zero, a row is equal to X, but shifted each time. Then you just sum up all rows (which are called ”partial products”.)

This is 4-bit binary multiplier. It takes 4-bit inputs and produces 8-bit output:

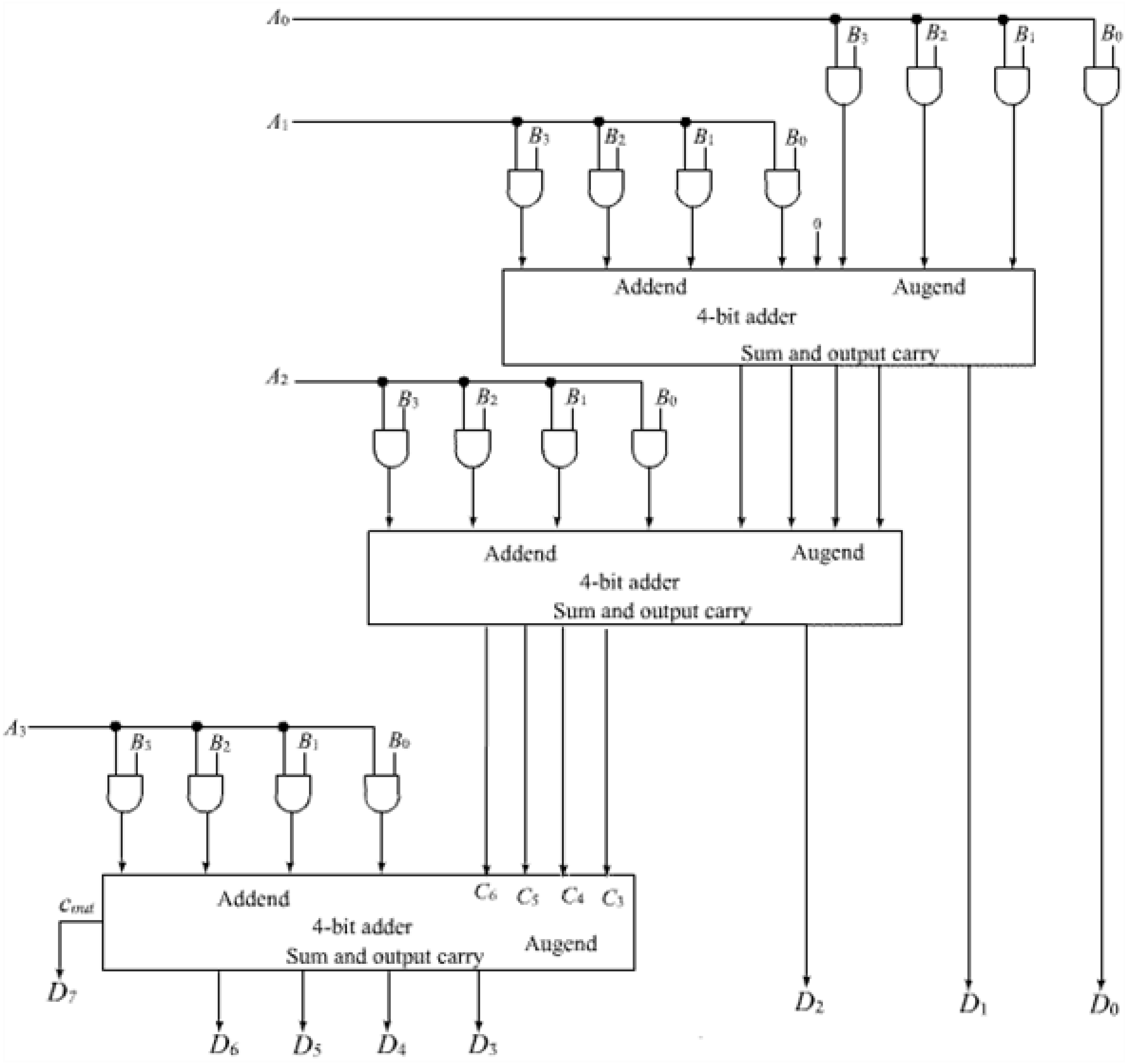


Figure 3.10: 4-bit binary multiplier

(The image has been taken from http://www.chegg.com/homework-help/binary-multiplier-multiplies-two-unsigned
)

I would build separate block, ”multiply by one bit” as a latch for each partial product:

```
    def AND_Tseitin(self, v1, v2, out):
        self.add_clause([self.neg(v1), self.neg(v2), out])
        self.add_clause([v1, self.neg(out)])
        self.add_clause([v2, self.neg(out)])

    def AND(self,v1, v2):
        out=self.create_var()
        self.AND_Tseitin(v1, v2, out)
        return out

...
```

```
    # bit is 0 or 1.
    # i.e., if it's 0, output is 0 (all bits)
    # if it's 1, output=input
    def mult_by_bit(self, X, bit):
        return [self.AND(i, bit) for i in X]

    # bit order: [MSB..LSB]
    # build multiplier using adders and mult_by_bit blocks:
    def multiplier(self, X, Y):
        assert len(X)==len(Y)
        out=[]
        #initial:
        prev=[self.const_false]*len(X)
        # first adder can be skipped, but I left thing "as is" to make it simpler
        for Y_bit in my_utils.rvr(Y):
            s, carry = self.adder(self.mult_by_bit(X, Y_bit), prev)
            out.append(s[-1])
            prev=[carry] + s[:-1]

        return prev + my_utils.rvr(out)
```

AND gate is constructed here using Tseitin transformations. This is quite popular way of encoding gates in CNF form, by adding additional variable: https://en.wikipedia.org/wiki/Tseytin_transformation. In fact, full-adder can be constructed without Mathematica, using logic gates, and encoded by Tseitin transformation.

3.13.3 As an electrical device

This is important thing to note: you can solder that multiplier as an electrical device, connect switches to inputs and LEDs to outputs, and it will multiply.

But an electrical device has inputs and inputs.

While SAT problem has just variables. Some of them are *grounded* to *false* or *true*, while others are like *dangling*. A SAT solver will find values for these *dangling* variables, if possible. (Or will report UNSAT if it's not possible.)

3.13.4 Gluing all together

```
#!/usr/bin/env python3

import itertools, subprocess, os, math, random
from operator import mul
import my_utils, SAT_lib
import functools

def factor(n):
    print ("factoring %d" % n)

    # size of inputs.
    # in other words, how many bits we have to allocate to store 'n'?
    input_bits=int(math.ceil(math.log(n,2)))
    print ("input_bits=%d" % input_bits)

    s=SAT_lib.SAT_lib(maxsat=False)

    factor1,factor2=s.alloc_BV(input_bits),s.alloc_BV(input_bits)
    product=s.multiplier(factor1,factor2)

    # at least one bit in each input must be set, except the lowest one.
    # hence we restrict inputs to be greater than 1
    if len(factor1)>1:
        s.fix(s.OR_list(factor1[:-1]), True)
    if len(factor2)>1:
        s.fix(s.OR_list(factor2[:-1]), True)
    # output has a size twice as bigger as each input:
    s.fix_BV(product, SAT_lib.n_to_BV(n,input_bits*2))

    if s.solve()==False:
        print ("%d is prime (unsat)" % n)
        return [n]

    # get inputs of multiplier:
    factor1_n=SAT_lib.BV_to_number(s.get_BV_from_solution(factor1))
    factor2_n=SAT_lib.BV_to_number(s.get_BV_from_solution(factor2))

    print ("factors of %d are %d and %d" % (n, factor1_n, factor2_n))
    # factor factors recursively:
    rt=sorted(factor (factor1_n) + factor (factor2_n))
    assert functools.reduce(mul, rt, 1)==n
    return rt

# infinite test:
def test():
    while True:
        print (factor (random.randrange(100000000000)))

#test()

print (factor(1234567890))
#print (factor(1073741827))
```

I just connect our number to output of multiplier and ask SAT solver to find inputs. If it's UNSAT, this is prime number. Then we factor factors recursively.

Also, we want block input factors of 1, because obviously, we do not interesting in the fact that n*1=n. I'm using wide OR gates for this.

Output:

```
 % python factor_SAT.py
factoring 1234567890
input_bits=31
factors of 1234567890 are 2 and 617283945
factoring 2
input_bits=1
2 is prime (unsat)
factoring 617283945
input_bits=30
factors of 617283945 are 3 and 205761315
factoring 3
input_bits=2
3 is prime (unsat)
factoring 205761315
input_bits=28
factors of 205761315 are 3 and 68587105
factoring 3
input_bits=2
3 is prime (unsat)
factoring 68587105
input_bits=27
factors of 68587105 are 5 and 13717421
factoring 5
input_bits=3
5 is prime (unsat)
factoring 13717421
input_bits=24
factors of 13717421 are 3607 and 3803
```

```
factoring 3607
input_bits=12
3607 is prime (unsat)
factoring 3803
input_bits=12
3803 is prime (unsat)
[2, 3, 3, 5, 3607, 3803]
```

So, $1234567890 = 2 \cdot 3 \cdot 3 \cdot 5 \cdot 3607 \cdot 3803$.

It works way faster than by Z3 solution, but still slow. It can factor numbers up to maybe ~2^{40}, while Wolfram Mathematica can factor ~2^{80} easily.

The full source code: https://smt.st/current_tree/equations/factor_SAT/factor_SAT.py.

3.13.5 Division using multiplier

Hard to believe, but why we couldn't define one of factors and ask SAT solver to find another factor? Then it will divide numbers! But, unfortunately, this is somewhat impractical, since it will work only if remainder is zero:

```
#!/usr/bin/env python3

import itertools, subprocess, os, math, random
from operator import mul
import my_utils, SAT_lib

def div(dividend,divisor):

    # size of inputs.
    # in other words, how many bits we have to allocate to store 'n'?
    input_bits=int(math.ceil(math.log(dividend,2)))
    print ("input_bits=%d" % input_bits)

    s=SAT_lib.SAT_lib(maxsat=False)

    factor1,factor2=s.alloc_BV(input_bits),s.alloc_BV(input_bits)
    product=s.multiplier(factor1,factor2)

    # connect divisor to one of multiplier's input:
    s.fix_BV(factor1, SAT_lib.n_to_BV(divisor,input_bits))
    # output has a size twice as bigger as each input.
    # connect dividend to multiplier's output:
    s.fix_BV(product, SAT_lib.n_to_BV(dividend,input_bits*2))

    if s.solve()==False:
        print ("remainder!=0 (unsat)")
        return None

    # get 2nd input of multiplier, which is quotient:
    return SAT_lib.BV_to_number(s.get_BV_from_solution(factor2))

print (div (1234567890123456789012345678 9*12345, 12345))
```

The full source code: https://smt.st/current_tree/equations/factor_SAT/div.py.

It works very fast, but still, slower than conventional ways.

3.13.6 Breaking RSA

It's not a problem to build multiplier with 4096 bit inputs and 8192 output, but it will not work in practice. Still, you can break toy-level demonstrational RSA problems with key less than 2^{40} or something like that (or larger, using Wolfram Mathematica).

3.13.7 Further reading

1, 2, 3.

3.14 Recalculating micro-spreadsheet using Z3Py

There is a nice exercise[15]: write a program to recalculate micro-spreadsheet, like this one:

```
1       0       B0+B2           A0*B0*C0
123     10      12              11
667     A0+B1   (C1*A0)*122     A3+C2
```

The result must be:

```
1       0       135     82041
123     10      12      11
667     11      1342    83383
```

As it turns out, though overkill, this can be solved using MK85 with little effort:

```
#!/usr/bin/python3

from MK85 import *
import sys, re

# MS Excel or LibreOffice style.
# first top-left cell is A0, not A1
def coord_to_name(R, C):
    return "ABCDEFGHIJKLMNOPQRSTUVWXYZ"[R]+str(C)

# open file and parse it as list of lists:
f=open(sys.argv[1],"r")
# filter(None, ...) to remove empty sublists:
ar=list(filter(None, [item.rstrip().split() for item in f.readlines()]))
f.close()

WIDTH=len(ar[0])
HEIGHT=len(ar)

s=MK85()

# cells{} is a dictionary with keys like "A0", "B9", etc:
cells={}
for R in range(HEIGHT):
    for C in range(WIDTH):
        name=coord_to_name(R, C)
        cells[name]=s.BitVec(name, 32)

cur_R=0
cur_C=0

for row in ar:
    for c in row:
        # string like "A0+B2" becomes "cells["A0"]+cells["B2"]":
        c=re.sub(r'([A-Z]{1}[0-9]+)', r'cells["\1"]', c)
        st="cells[\"%s\"]==%s" % (coord_to_name(cur_R, cur_C), c)
        # evaluate string. Z3Py expression is constructed at this step:
        e=eval(st)
        # add constraint:
        s.add (e)
        cur_C=cur_C+1
    cur_R=cur_R+1
    cur_C=0

if s.check():
    m=s.model()
```

[15] The blog post in Russian: http://thesz.livejournal.com/280784.html

```
    for r in range(HEIGHT):
        for c in range(WIDTH):
            sys.stdout.write (str(m[coord_to_name(r, c)])+"\t")
        sys.stdout.write ("\n")

else:
    print ("UNSAT")
```

(https://smt.st/current_tree/equations/spreadsheet/spreadsheet_MK85.py)

All we do is just creating pack of variables for each cell, named A0, B1, etc, of integer type. All of them are stored in *cells[]* dictionary. Key is a string. Then we parse all the strings from cells, and add to list of constraints *A0=123* (in case of number in cell) or *A0=B1+C2* (in case of expression in cell). There is a slight preparation: string like *A0+B2* becomes *cells[”A0”]+cells[”B2”]*.

Then the string is evaluated using Python *eval()* method, which is highly dangerous [16]: imagine if end-user could add a string to cell other than expression? Nevertheless, it serves our purposes well, because this is a simplest way to pass a string with expression into MK85.

3.14.1 Z3

The source code almost the same:

```
#!/usr/bin/python3

from z3 import *
import sys, re

# MS Excel or LibreOffice style.
# first top-left cell is A0, not A1
def coord_to_name(R, C):
    return "ABCDEFGHIJKLMNOPQRSTUVWXYZ"[R]+str(C)

# open file and parse it as list of lists:
f=open(sys.argv[1],"r")
# filter(None, ...) to remove empty sublists:
ar=list(filter(None, [item.rstrip().split() for item in f.readlines()]))
f.close()

WIDTH=len(ar[0])
HEIGHT=len(ar)

# cells{} is a dictionary with keys like "A0", "B9", etc:
cells={}
for R in range(HEIGHT):
    for C in range(WIDTH):
        name=coord_to_name(R, C)
        cells[name]=Int(name)

s=Solver()

cur_R=0
cur_C=0

for row in ar:
    for c in row:
        # string like "A0+B2" becomes "cells["A0"]+cells["B2"]":
        c=re.sub(r'([A-Z]{1}[0-9]+)', r'cells["\1"]', c)
        st="cells[\"%s\"]==%s" % (coord_to_name(cur_R, cur_C), c)
        # evaluate string. Z3Py expression is constructed at this step:
        e=eval(st)
        # add constraint:
```

[16] http://stackoverflow.com/questions/1832940/is-using-eval-in-python-a-bad-practice

```
        s.add (e)
        cur_C=cur_C+1
    cur_R=cur_R+1
    cur_C=0

result=s.check()
print (result)
if result==sat:
    m=s.model()
    for r in range(HEIGHT):
        for c in range(WIDTH):
            sys.stdout.write (str(m[cells[coord_to_name(r, c)]])+"\t")
        sys.stdout.write ("\n")
```

(https://smt.st/current_tree/equations/spreadsheet/spreadsheet_Z3_1.py)

3.14.2 Unsat core

Now the problem: what if there is circular dependency? Like:

```
1       0       B0+B2           A0*B0
123     10      12              11
C1+123  C0*123  A0*122          A3+C2
```

Two first cells of the last row (C0 and C1) are linked to each other. Our program will just tells “unsat”, meaning, it couldn’t satisfy all constraints together. We can’t use this as error message reported to end-user, because it’s highly unfriendly.

Our question now is: which variables ”spoiling” the whole spreadsheet and ”to be dropped” to get everything solvable?

We can fetch *unsat core*, i.e., list of variables which Z3 finds conflicting.

```
...
s=Solver()
s.set(unsat_core=True)
...
        # add constraint:
        s.assert_and_track(e, coord_to_name(cur_R, cur_C))
...
if result=="sat":
...
else:
    print s.unsat_core()
```

(https://smt.st/current_tree/equations/spreadsheet/spreadsheet_Z3_2.py)

We must explicitly turn on unsat core support and use *assert_ and_ track()* instead of *add()* method, because this feature slows down the whole process, and is turned off by default. That works:

```
 % python 2.py test_circular
unsat
[C0, C1]
```

Perhaps, these variables could be removed from the 2D array, marked as *unresolved* and the whole spreadsheet could be recalculated again.

... but. Needless to say, that this is a circular dependency, but not a problem (`C1+123` and `C0-123`):

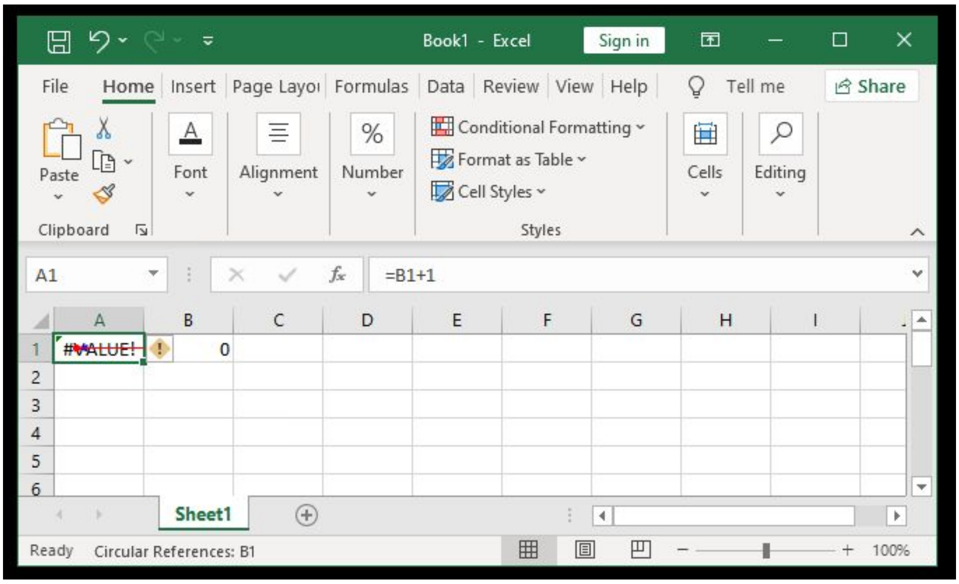
```
1       0       B0+B2           A0*B0
123     10      12              11
C1+123  C0-123  A0*122          A3+C2
```

My solvers can find at least two (correct) solutions:

```
sat
1       0       135     123
123     10      12      11
0       -123    122     245
```

```
sat
1         0         135       123
123       10        12        11
-1        -124      122       245
```

Funny, but Microsoft Excel can't handle such circular (but correct) dependencies:



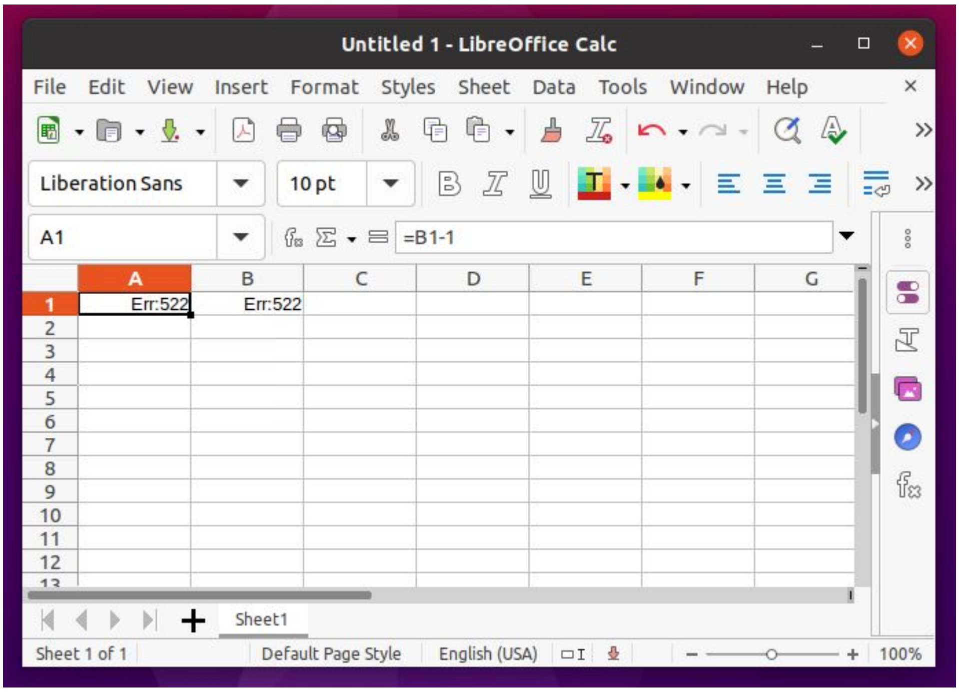
A1: =B1+1

	A	B
1	#VALUE!	0

Circular References: B1

Figure 3.11: Microsoft Excel

What about LibreOffice Calc?



A1: =B1-1

	A	B
1	Err:522	Err:522

Figure 3.12: LibreOffice Calc

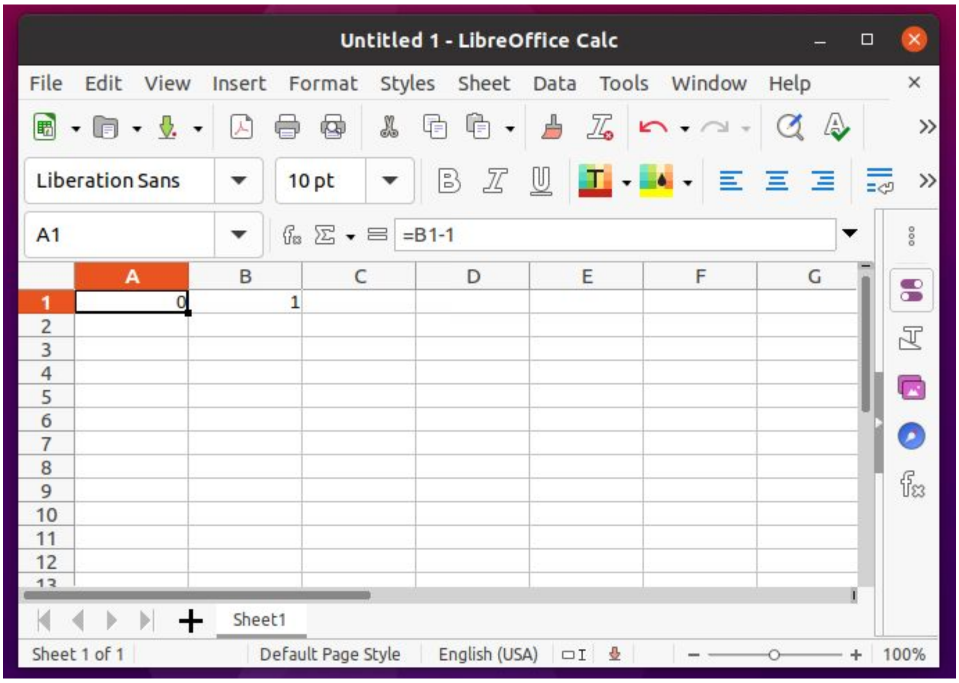
Error 522 is[17]: "Circular reference", "Formula refers directly or indirectly to itself and the Iterations option is not set under Tools - Options - LibreOffice Calc - Calculate.".

OK, I set this option on (to 0) and got Error 523:

"The calculation procedure does not converge", "Function missed a targeted value, or iterative references do not reach the minimum change within the maximum number of steps set."

[17] https://help.libreoffice.org/6.4/en-GB/text/scalc/05/02140000.html

Setting *Iterations* option to 1 seems to solve that problem:



Untitled 1 - LibreOffice Calc
File Edit View Insert Format Styles Sheet Data Tools Window Help
Liberation Sans
10 pt
Sheet1
Sheet 1 of 1
Default Page Style
English (USA)
100%

A1 =B1-1

	A	B	C	D	E	F	G
1	0	1					

Figure 3.13: LibreOffice Calc

3.14.3 Stress test

How to generate a large random spreadsheet? What we can do. First, create random DAG[18], like this one:

[18] Directed acyclic graph

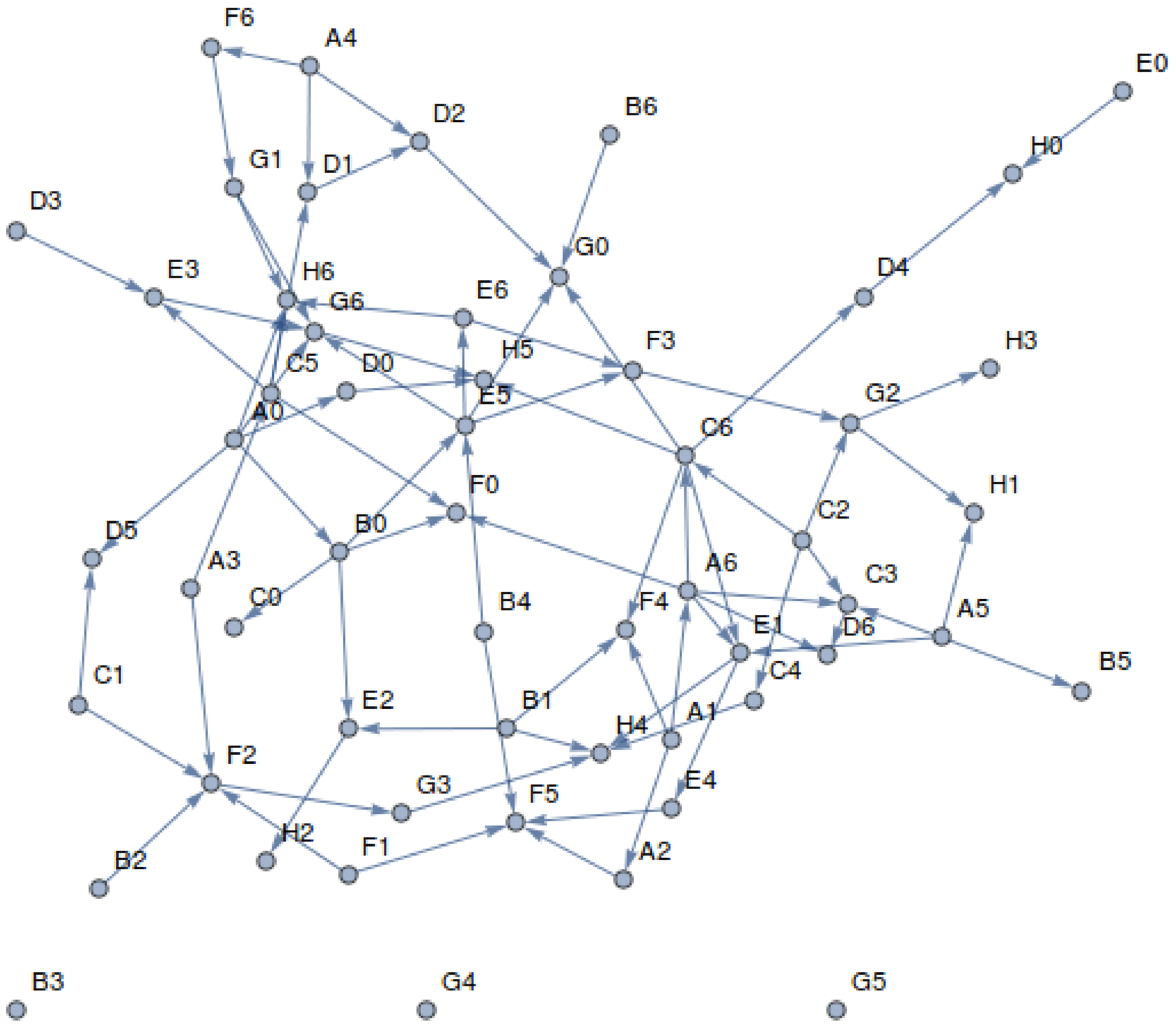


Figure 3.14: Random DAG

Arrows will represent information flow. So a vertex (node) which has no incoming arrows to it (indegree=0), can be set to a random number. Then we use topological sort to find dependencies between vertices. Then we assign spreadsheet cell names to each vertex. Then we generate random expression with random operations/numbers/cells to each cell, with the use of information from topological sorted graph.

```
(* Utility functions *)
In[1]:= findSublistBeforeElementByValue[lst_,element_]:=lst[[ 1;;Position[lst,
    element][[1]][[1]]-1]]

(* Input in 1..∞ range. 1->A0, 2->A1, etc *)
In[2]:= vertexToName[x_,width_]:=StringJoin[FromCharacterCode[ToCharacterCode["A
    "][[1]]+Floor[(x-1)/width]],ToString[Mod[(x-1),width]]]

In[3]:= randomNumberAsString[]:=ToString[RandomInteger[{1,1000}]]

In[4]:= interleaveListWithRandomNumbersAsStrings[lst_]:=Riffle[lst,Table[
    randomNumberAsString[],Length[lst]-1]]

(* We omit division operation because micro-spreadsheet evaluator can't handle
    division by zero *)
In[5]:= interleaveListWithRandomOperationsAsStrings[lst_]:=Riffle[lst,Table[
   RandomChoice[{"+","-","*"}],Length[lst]-1]]

In[6]:= randomNonNumberExpression[g_,vertex_]:=StringJoin[
   interleaveListWithRandomOperationsAsStrings[
   interleaveListWithRandomNumbersAsStrings[Map[vertexToName[#,WIDTH]&,
   pickRandomNonDependentVertices[g,vertex]]]]]

In[7]:= pickRandomNonDependentVertices[g_,vertex_]:=DeleteDuplicates[RandomChoice[
   findSublistBeforeElementByValue[TopologicalSort[g],vertex],RandomInteger[{1,5}]]]

In[8]:= assignNumberOrExpr[g_,vertex_]:=If[VertexInDegree[g,vertex]==0,
   randomNumberAsString[],randomNonNumberExpression[g,vertex]]

(* Main part *)
(* Create random graph *)
In[21]:= WIDTH=7;HEIGHT=8;TOTAL=WIDTH*HEIGHT
Out[21]= 56

In[24]:= g=DirectedGraph[RandomGraph[BernoulliGraphDistribution[TOTAL,0.05]],"Acyclic
   "];

...

(* Generate random expressions and numbers *)
In[26]:= expressions=Map[assignNumberOrExpr[g,#]&,VertexList[g]];

(* Make 2D table of it *)
In[27]:= t=Partition[expressions,WIDTH];

(* Export as tab-separated values *)
In[28]:= Export["/home/dennis/1.txt",t,"TSV"]
Out[28]= /home/dennis/1.txt

In[29]:= Grid[t,Frame->All,Alignment->Left]
```

Here is an output from *Grid[]*:

846	499	A3*913-H4	...	...	...	...
B4*860+D2	999	59	...	...	...	...
G6*379-C3-436-C4-289+H6	972	804	...	...	...	...
F2	E0	B6-731-D3+791+B4*92+C1	...	...	...	...
519	G1*402+D1*107*G3-458*A1	D3	...	...	...	...
F5-531+B5-222*E4	9	B5+106*B6+600-B1	...	...	...	...
C3-956*A5	G4*408-D3*290*B6-899*G5+400+F1	B2-701+H6	...	...	...	..
B4-792*H4*407+F6-425-E1	D2	D3	...	...	...	...

Using this script, I can generate random spreadsheet of $26 \cdot 500 = 13000$ cells, which seems to be processed by Z3 in couple of seconds.

3.14.4 The files

The files, including Mathematica notebook: https://smt.st/current_tree/equations/spreadsheet.

3.15 Discrete tomography

The following puzzle can be solved using SAT/SMT solvers without effort, you can do this as a homework:

Black Box is an abstract board game for one or two players, which simulates shooting rays into a black box to deduce the locations of ”atoms” hidden inside. It was created by Eric Solomon. The board game was published by Waddingtons from the mid-1970s and by Parker Brothers in the late 1970s. The game can also be played with pen and paper, and there are numerous computer implementations for many different platforms, including one which can be run from the Emacs text editor.

Black Box was inspired by the work of Godfrey Hounsfield who was awarded the 1979 Nobel Prize in Medicine for his invention of the CAT scanner.

(https://en.wikipedia.org/wiki/Black_Box_(game))

How computed tomography (CT scan) actually works? A human body is bombarded by X-rays in various angles by X-ray tube in rotating torus. X-ray detectors are also located in torus, and all the information is recorded.

Many of readers probably have seen CT scan device in dental clinics, like:

Figure 3.15: From the https://endocare.ca/ website

Video: https://youtu.be/0eoYdtAmr8s?t=114. It rotates around your head and then returns to initial position. One half of device has something emitting (so far, no matter what is this: X-ray or MRI), another half has an array of detectors.

Here is we can simulate a simple tomograph. An “i” character is rotating and will be “enlighten” at 4 angles. Let’s imagine, character is bombarded by X-ray tube at left. All asterisks in each row is then summed and sum is ”received” by the X-ray detector at the right.

```
WIDTH= 11 HEIGHT= 11
angleπ=(/4)*0
    **      2
    **      2
            0
   ***      3
    **      2
    **      2
    **      2
    **      2
    **      2
   ****     4
            0
[2, 2, 0, 3, 2, 2, 2, 2, 2, 4, 0] ,
angleπ=(/4)*1
            0
            0
```

```
  *         1
 **         2
    *       1
    **      2
     **     2
     ****   4
       *    1
      *     1
            0
[0, 0, 1, 2, 1, 2, 2, 4, 1, 1, 0] ,
angleπ=(/4)*2
            0
            0
            0
            0
         *  1
** *******  9
** *******  9
   *     *  2
            0
            0
            0
[0, 0, 0, 0, 1, 9, 9, 2, 0, 0, 0] ,
angleπ=(/4)*3
            0
            0
       *    1
       **   2
      ** *  3
     ***    3
    **      2
            0
  **        2
   *        1
            0
[0, 0, 1, 2, 3, 3, 2, 0, 2, 1, 0] ,
```

(The source code: https://smt.st/current_tree/equations/tomo/gen.py)

All we got from our toy-level tomograph is 4 vectors, these are sums of all asterisks in rows for 4 angles:

```
[2, 2, 0, 3, 2, 2, 2, 2, 2, 4, 0] ,
[0, 0, 1, 2, 1, 2, 2, 4, 1, 1, 0] ,
[0, 0, 0, 0, 1, 9, 9, 2, 0, 0, 0] ,
[0, 0, 1, 2, 3, 3, 2, 0, 2, 1, 0] ,
```

How do we recover the original image? We are going to represent 11*11 matrix, where sum of each row must be equal to some value we already know. Then we rotate matrix, and do this again.

For the first matrix, the system of equations looks like that (we put there a values from the first vector):

```
C1,1 + C1,2 + C1,3 + ... + C1,11 =      2
C2,1 + C2,2 + C2,3 + ... + C2,11 =      2

...

C10,1 + C10,2 + C10,3 + ... + C10,11 =  4
C11,1 + C11,2 + C11,3 + ... + C11,11 =  0
```

We also build similar systems of equations for each angle.

The “rotate” function has been taken from the generation program, because, due to the nature of Python’s dynamic type system. it’s not important for the function to what operate on: strings, characters, or Z3 variable instances, so it works very well for all of them.

```
#!/usr/bin/env python3
```

```
#-*- coding: utf-8 -*-

import math, sys
from z3 import *

# https://en.wikipedia.org/wiki/Rotation_matrix
def rotate(pic, angle):
    WIDTH=len(pic[0])
    HEIGHT=len(pic)
    #print WIDTH, HEIGHT
    assert WIDTH==HEIGHT
    ofs=int(WIDTH/2)

    out = [[0 for x in range(WIDTH)] for y in range(HEIGHT)]

    for x in range(-ofs,ofs):
        for y in range(-ofs,ofs):
            newX = int(round(math.cos(angle)*x - math.sin(angle)*y,3))+ofs
            newY = int(round(math.sin(angle)*x + math.cos(angle)*y,3))+ofs
            # clip at boundaries, hence min(..., HEIGHT-1)
            out[min(newX,HEIGHT-1)][min(newY,WIDTH-1)]=pic[x+ofs][y+ofs]
    return out

vectors=[
[2, 2, 0, 3, 2, 2, 2, 2, 2, 4, 0] ,
[0, 0, 1, 2, 1, 2, 2, 4, 1, 1, 0] ,
[0, 0, 0, 0, 1, 9, 9, 2, 0, 0, 0] ,
[0, 0, 1, 2, 3, 3, 2, 0, 2, 1, 0]]

WIDTH = HEIGHT = len(vectors[0])

s=Solver()
cells=[[Int('cell_r=%d_c=%d' % (r,c)) for c in range(WIDTH)] for r in range(HEIGHT)]

# monochrome picture, only 0's or 1's:
for c in range(WIDTH):
    for r in range(HEIGHT):
        s.add(Or(cells[r][c]==0, cells[r][c]==1))

ANGLES=len(vectors)
for a in range(ANGLES):
    angle=a*(math.pi/ANGLES)
    rows=rotate(cells, angle)
    r=0
    for row in rows:
        # sum of row must be equal to the corresponding element of vector:
        s.add(Sum(*row)==vectors[a][r])
        r=r+1

print (s.check())
m=s.model()
for r in range(HEIGHT):
    for c in range(WIDTH):
        if str(m[cells[r][c]])=="1":
            sys.stdout.write("*")
        else:
            sys.stdout.write(" ")
    print ("")
```

(The source code: https://smt.st/current_tree/equations/tomo/solve.py)

That works:

```
% python solve.py
sat
    **
    **

   ***
    **
    **
    **
    **
    **
   ****
```

In other words, all SMT-solver does here is solving a system of equations.

So, 4 angles are enough. What if we could use only 3 angles?

```
WIDTH= 11 HEIGHT= 11
angleπ=(/3)*0
    **      2
    **      2
            0
   ***      3
    **      2
    **      2
    **      2
    **      2
    **      2
   ****     4
            0
[2, 2, 0, 3, 2, 2, 2, 2, 2, 4, 0] ,
angleπ=(/3)*1
            0
            0
            0
 **         2
 **         2
   ***      3
     ****   4
       **   2
       *    1
            0
            0
[0, 0, 0, 2, 2, 3, 4, 2, 1, 0, 0] ,
angleπ=(/3)*2
            0
            0
            0
       **   2
       **   2
     *****  5
    **      2
 **         2
  *         1
            0
            0
[0, 0, 0, 2, 2, 5, 2, 2, 1, 0, 0] ,
```

No, it's not enough:

```
% time python solve3.py
sat
 *  *
    **
```

```
     * **
   **
   *  *
    **
     *   *
*   *
   ****
```

However, the result is correct, but only 3 vectors allows too many possible “original images”, and Z3 SMT-solver finds first.

Further reading: https://en.wikipedia.org/wiki/Discrete_tomography, https://en.wikipedia.org/wiki/2-satisfiability#Discrete_tomography, https://en.wikipedia.org/wiki/Black_Box_(game).

3.16 Cribbage

I’ve found this problem in the Ronald L. Graham, Donald E. Knuth, Oren Patashnik – “Concrete Mathematics” book:

> Cribbage players have long been aware that $15 = 7 + 8 = 4 + 5 + 6 = 1 + 2 + 3 + 4 + 5$. Find the number of ways to represent 1050 as a sum of consecutive positive integers. (The trivial representation ‘1050’ by itself counts as one way; thus there are four, not three, ways to represent 15 as a sum of consecutive positive integers. Incidentally, a knowledge of cribbage rules is of no use in this problem.)

My solution:

```
#!/usr/bin/env python3

from z3 import *

def attempt(terms, N):
    #print "terms = %d" % terms

    cells=[Int('%d' % i) for i in range(terms)]

    s=Solver()

    for i in range(terms-1):
       s.add(cells[i]+1 == cells[i+1])

    s.add(Sum(cells)==N)

    s.add(cells[0]>0)

    if s.check()==sat:
        m=s.model()
        print ("(%d terms) %d + ... + %d == %d" % (terms, m[cells[0]].as_long(), m[
            cells[terms-1]].as_long(), N))

#N=15
N=1050

for i in range(2,N):
    attempt(i, N)
```

The result:

```
(3 terms) 349 + ... + 351 == 1050
(4 terms) 261 + ... + 264 == 1050
(5 terms) 208 + ... + 212 == 1050
```

```
(7 terms) 147 + ... + 153 == 1050
(12 terms) 82 + ... + 93 == 1050
(15 terms) 63 + ... + 77 == 1050
(20 terms) 43 + ... + 62 == 1050
(21 terms) 40 + ... + 60 == 1050
(25 terms) 30 + ... + 54 == 1050
(28 terms) 24 + ... + 51 == 1050
(35 terms) 13 + ... + 47 == 1050
```

3.17 Solving Problem Euler 31: "Coin sums"

> In the United Kingdom the currency is made up of pound (£) and pence (p). There are eight coins in general circulation:
>
> 1p, 2p, 5p, 10p, 20p, 50p, £1 (100p), and £2 (200p).
>
> It is possible to make £2 in the following way:
>
> 1×£1 + 1×50p + 2×20p + 1×5p + 1×2p + 3×1p
>
> How many different ways can £2 be made using any number of coins?

(Problem Euler 31 — Coin sums)

Using Z3 for solving this is overkill, and also slow, but nevertheless, it works, showing all possible solutions as well. The piece of code for blocking already found solution and search for next, and thus, counting all solutions, was taken from Stack Overflow answer [19]. This is also called "model counting". Constraints like "a>=0" must be present, because Z3 solver will find solutions with negative numbers.

```
#!/usr/bin/python3

from z3 import *

a,b,c,d,e,f,g,h = Ints('a b c d e f g h')
s = Solver()
s.add(1*a + 2*b + 5*c + 10*d + 20*e + 50*f + 100*g + 200*h == 200,
   a>=0, b>=0, c>=0, d>=0, e>=0, f>=0, g>=0, h>=0)
result=[]

while True:
    if s.check() == sat:
        m = s.model()
        print (m)
        result.append(m)
        # Create a new constraint the blocks the current model
        block = []
        for d in m:
            # create a constant from declaration
            c=d()
            block.append(c != m[d])
        s.add(Or(block))
    else:
        print (len(result))
        break
```

Works very slow, and this is what it produces:

```
[h = 0, g = 0, f = 0, e = 0, d = 0, c = 0, b = 0, a = 200]
[f = 1, b = 5, a = 0, d = 1, g = 1, h = 0, c = 2, e = 1]
[f = 0, b = 1, a = 153, d = 0, g = 0, h = 0, c = 1, e = 2]
```

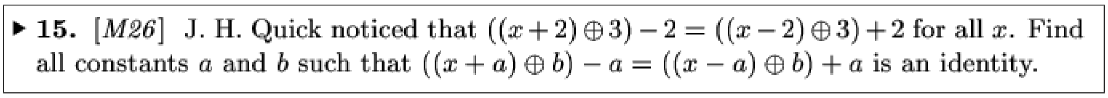
[19] http://stackoverflow.com/questions/11867611/z3py-checking-all-solutions-for-equation, another question: http://stackoverflow.com/questions/13395391/z3-finding-all-satisfying-models

```
...
[f = 0, b = 31, a = 33, d = 2, g = 0, h = 0, c = 17, e = 0]
[f = 0, b = 30, a = 35, d = 2, g = 0, h = 0, c = 17, e = 0]
[f = 0, b = 5, a = 50, d = 2, g = 0, h = 0, c = 24, e = 0]
```

73682 results in total.

3.18 Exercise 15 from TAOCP “7.1.3 Bitwise tricks and techniques”

Page 53 from the fasc1a.ps, or: http://www.cs.utsa.edu/~wagner/knuth/fasc1a.pdf



▶ **15.** [*M26*] J. H. Quick noticed that $((x+2)\oplus 3)-2=((x-2)\oplus 3)+2$ for all x. Find all constants a and b such that $((x+a)\oplus b)-a=((x-a)\oplus b)+a$ is an identity.

Figure 3.16: Page 53

Soltuion:

```
#!/usr/bin/env python3

from z3 import *

s=Solver()

a, b=BitVecs('a b', 4)
x, y=BitVecs('x y', 4)

s.add(ForAll(x, ForAll(y,  ((x+a)^b)-a == ((x-a)^b)+a  )))

# enumerate all possible solutions:
results=[]
while True:
    if s.check() == sat:
        m = s.model()
        print (m)

        results.append(m)
        block = []
        for d in m:
            c=d()
            block.append(c != m[d])
        s.add(Or(block))
    else:
        print ("results total=", len(results))
        break
```

For 4-bit bitvectors:

```

...

[b = 7, a = 0]
[b = 6, a = 8]
[b = 7, a = 8]
[b = 6, a = 12]
[b = 7, a = 12]
[b = 12, a = 0]
[b = 13, a = 0]
[b = 12, a = 8]
[b = 13, a = 8]
[b = 12, a = 4]
```

```
[b = 13, a = 4]
[b = 12, a = 12]
[b = 13, a = 12]
[b = 14, a = 0]
[b = 15, a = 0]
[b = 14, a = 4]
[b = 15, a = 4]
[b = 14, a = 8]
[b = 15, a = 8]
[b = 14, a = 12]
[b = 15, a = 12]
results total= 128
```

3.19 Generating de Bruijn sequences using SMT solver

(Untitled collection of math notes[20]has a part about de Bruijn sequences.)

The following piece of quite esoteric code calculates number of leading zero bits [21]:

```
int v[64]=
        { -1,31, 8,30, -1, 7,-1,-1, 29,-1,26, 6, -1,-1, 2,-1,
          -1,28,-1,-1, -1,19,25,-1, 5,-1,17,-1, 23,14, 1,-1,
           9,-1,-1,-1, 27,-1, 3,-1, -1,-1,20,-1, 18,24,15,10,
          -1,-1, 4,-1, 21,-1,16,11, -1,22,-1,12, 13,-1, 0,-1 };

int LZCNT(uint32_t x)
{
    x |= x >> 1;
    x |= x >> 2;
    x |= x >> 4;
    x |= x >> 8;
    x |= x >> 16;
    x *= 0x4badf0d;
    return v[x >> 26];
}
```

(This is usually done using simpler algorithm, but it will contain conditional jumps, which is bad for CPUs starting at RISC. There are no conditional jumps in this algorithm.)

The magic number used here is called *de Bruijn sequence*, and I once used bruteforce to find it (one of the results was *0x4badf0d*, which is used here). But what if we need magic number for 64-bit values? Bruteforce is not an option here.

If you already read about these sequences in my blog or in other sources, you can see that the 32-bit magic number is a number consisting of 5-bit overlapping chunks, and all chunks must be unique, i.e., must not be repeating.

For 64-bit magic number, these are 6-bit overlapping chunks.

To find the magic number, one can find a Hamiltonian path of a de Bruijn graph. But I've found that Z3 is also can do this, though, overkill, but this is more illustrative.

Listing 3.12: Z3Py program

```
#!/usr/bin/python3

from z3 import *

out = BitVec('out', 64)

tmp=[]
for i in range(64):
    tmp.append((out>>i)&0x3F)

s=Solver()
```

[20] https://xn--mxacd.xn--qxam/math-notes.pdf

[21] https://en.wikipedia.org/wiki/Find_first_set

```
# all overlapping 6-bit chunks must be distinct:
s.add(Distinct(*tmp))
# MSB must be zero:
s.add((out&0x8000000000000000)==0)

print (s.check())

result=s.model()[out].as_long()
print ("0x%x" % result)

# print overlapping 6-bit chunks:
for i in range(64):
    t=(result>>i)&0x3F
    print (" "*(63-i) + format(t, 'b').zfill(6))
```

We just enumerate all overlapping 6-bit chunks and tell Z3 that they must be unique (see `Distinct`).

Listing 3.13: The output

```
sat
0x79c52dd0991abf60
                                                               100000
                                                              110000
                                                             011000
                                                            101100
                                                           110110
                                                          111011
                                                         111101
                                                        111110
                                                       111111
                                                      011111
                                                     101111
                                                    010111
                                                   101011
                                                  010101
                                                 101010
                                                110101
                                               011010
                                              001101
                                             000110
                                            100011
                                           010001
                                          001000
                                         100100
                                        110010
                                       011001
                                      001100
                                     100110
                                    010011
                                   001001
                                  000100
                                 000010
                                100001
                               010000
                              101000
                             110100
                            111010
                           011101
                          101110
                         110111
                        011011
```

```
                       101101
                      010110
                     001011
                    100101
                   010010
                  101001
                 010100
                001010
               000101
              100010
             110001
            111000
           011100
          001110
         100111
        110011
       111001
      111100
     011110
    001111
   000111
  000011
 000001
000000
```

Overlapping chunks are clearly visible. So the magic number is *0x79c52dd0991abf60*. Let's check:

```
#include <stdint.h>
#include <stdio.h>
#include <assert.h>

#define MAGIC 0x79c52dd0991abf60

int magic_tbl[64];

// returns single bit position counting from LSB
// not works for i==0
int bitpos (uint64_t i)
{
        return magic_tbl[(MAGIC/i) & 0x3F];
};

// count trailing zeroes
// not works for i==0
int tzcnt (uint64_t i)
{
        uint64_t a=i & (-i);
        return magic_tbl[(MAGIC/a) & 0x3F];
};

int main()
{
        // construct magic table
        // may be omitted in production code
        for (int i=0; i<64; i++)
                magic_tbl[(MAGIC/(1ULL<<i)) & 0x3F]=i;

        // test
        for (int i=0; i<64; i++)
        {
                printf ("input=0x%llx, result=%d\n", 1ULL<<i, bitpos (1ULL<<i));
                assert(bitpos(1ULL<<i)==i);
        };
        assert(tzcnt (0xFFFF0000)==16);
        assert(tzcnt (0xFFFF0010)==4);
};
```

That works!

The problem is easy enough to be tackled MK85, although, it's not as fast as Z3, of course: src, output.

3.20 Solving the $x^y = 19487171$ equation

Find x,y for $x^y = 19487171$. The correct result x=11, y=7. It's like http://reference.wolfram.com/language/ref/Surd.html.

The non-standard function `bvmul_no_overflow` is used here. it behaves like bvmul, but high part is forced to be zero. This is not like most programming languages and CPUs do multiplication (the result there is modulo 2^n, where n is width of CPU register). However, thus it's simpler for me to write this all without adding additional `zero_extend` function.

```
; tested with MK85

(set-logic QF_BV)
(declare-fun x () (_ BitVec 32))
(declare-fun y () (_ BitVec 4))
(declare-fun out () (_ BitVec 32))

; like switch() or if() tree:
(assert (= out
        (ite (= y #x2) (bvmul_no_overflow x x)
        (ite (= y #x3) (bvmul_no_overflow x x x)
        (ite (= y #x4) (bvmul_no_overflow x x x x)
        (ite (= y #x5) (bvmul_no_overflow x x x x x)
        (ite (= y #x6) (bvmul_no_overflow x x x x x x)
        (ite (= y #x7) (bvmul_no_overflow x x x x x x x)
        (_ bv0 32)))))))))

(assert (= out (_ bv19487171 32)))

(check-sat)
(get-model)
```

Listing 3.14: The solution

```
(model
        (define-fun x () (_ BitVec 32) (_ bv11 32)) ; 0xb
        (define-fun y () (_ BitVec 4) (_ bv7 4)) ; 0x7
        (define-fun out () (_ BitVec 32) (_ bv19487171 32)) ; 0x12959c3
)
```

It is important to note that MK85 has no idea about Newton's method of finding square/cubic/etc roots...

3.21 Exercise

As an exercise, try to encode this problem using SMT-LIB 2.0 or Z3Py API:

> If a merchant buys 138 yards of cloth, some of which is black and some blue, for 540 roubles, how many yards of each did he buy if the blue cloth cost 5 roubles a yard and the black cloth 3?

(Anton Chekhov - The Tutor[22].)

[22] English translation, Russian original

Chapter 4

Proofs

SAT/SMT solvers can't prove *correctness* of something, or if the model behaves as the author wanted.

However, it can prove equivalence of two expressions or models.

4.1 Using Z3 theorem prover to prove equivalence of some weird alternative to XOR operation

(The test was first published in my blog at April 2015: http://blog.yurichev.com/node/86).

There is a "A Hacker's Assistant" program[1] (*Aha!*) written by Henry Warren, who is also the author of the great "Hacker's Delight" book.

The *Aha!* program is essentially *superoptimizer*[2], which blindly brute-force a list of some generic RISC CPU instructions to achieve shortest possible (and jumpless or branch-free) CPU code sequence for desired operation. For example, *Aha!* can find jumpless version of abs() function easily.

Compiler developers use superoptimization to find shortest possible (and/or jumpless) code, but I tried to do otherwise—to find longest code for some primitive operation. I tried *Aha!* to find equivalent of basic XOR operation without usage of the actual XOR instruction, and the most bizarre example *Aha!* gave is:

```
Found a 4-operation program:
   add   r1,ry,rx
   and   r2,ry,rx
   mul   r3,r2,-2
   add   r4,r3,r1
   Expr: (((y & x)*-2) + (y + x))
```

And it's hard to say, why/where we can use it, maybe for obfuscation, I'm not sure. I would call this *suboptimization* (as opposed to *superoptimization*). Or maybe *superdeoptimization*.

But my another question was also, is it possible to prove that this is correct formula at all? The *Aha!* checking some input/output values against XOR operation, but of course, not all the possible values. It is 32-bit code, so it may take very long time to try all possible 32-bit inputs to test it.

We can try Z3 theorem prover for the job. It's called *prover*, after all.

So I wrote this:

```
#!/usr/bin/python
from z3 import *

x = BitVec('x', 32)
y = BitVec('y', 32)
output = BitVec('output', 32)
s = Solver()
s.add(x^y==output)
s.add(((y & x)*0xFFFFFFFE) + (y + x)!=output)
print s.check()
```

In plain English language, this means "are there any case for x and y where $x \oplus y$ doesn't equals to $((y\&x)*-2)+(y+x)$?" ...and Z3 prints "unsat", meaning, it can't find any counterexample to the equation. So this *Aha!* result is proved to be working just like XOR operation.

[1] http://www.hackersdelight.org/

[2] http://en.wikipedia.org/wiki/Superoptimization

Oh, I also tried to extend the formula to 64 bit:

```
#!/usr/bin/python
from z3 import *

x = BitVec('x', 64)
y = BitVec('y', 64)
output = BitVec('output', 64)
s = Solver()
s.add(x^y==output)
s.add(((y & x)*0xFFFFFFFE) + (y + x)!=output)
print s.check()
```

Nope, now it says "sat", meaning, Z3 found at least one counterexample. Oops, it's because I forgot to extend -2 number to 64-bit value:

```
#!/usr/bin/python
from z3 import *

x = BitVec('x', 64)
y = BitVec('y', 64)
output = BitVec('output', 64)
s = Solver()
s.add(x^y==output)
s.add(((y & x)*0xFFFFFFFFFFFFFFFE) + (y + x)!=output)
print s.check()
```

Now it says "unsat", so the formula given by *Aha!* works for 64-bit code as well.

4.1.1 In SMT-LIB form

Now we can rephrase our expression to more suitable form: $(x + y - ((x \& y) << 1))$. It also works well in Z3Py:

```
#!/usr/bin/python
from z3 import *

x = BitVec('x', 64)
y = BitVec('y', 64)
output = BitVec('output', 64)
s = Solver()
s.add(x^y==output)
s.add((x + y - ((x & y)<<1)) != output)
print s.check()
```

Here is how to define it in SMT-LIB way:

```
(declare-const x (_ BitVec 64))
(declare-const y (_ BitVec 64))
(assert
        (not
                (=
                        (bvsub
                                (bvadd x y)
                                (bvshl (bvand x y) (_ bv1 64)))
                        (bvxor x y)
                )
        )
)
(check-sat)
```

4.1.2 Using universal quantifier

Z3 supports universal quantifier `exists`, which is true if at least one set of variables satistfied underlying condition:

```
(declare-const x (_ BitVec 64))
(declare-const y (_ BitVec 64))
(assert
        (exists ((x (_ BitVec 64)) (y (_ BitVec 64)))
                (not (=
                        (bvsub
                                (bvadd x y)
                                (bvshl (bvand x y) (_ bv1 64))
                        )
                        (bvxor x y)
                ))
        )
)
(check-sat)
```

It returns “unsat”, meaning, Z3 couldn’t find any counterexample of the equation, i.e., it’s not exist.

This is also known as $\exists$ in mathematical logic lingo.

Z3 also supports universal quantifier `forall`, which is true if the equation is true for all possible values. So we can rewrite our SMT-LIB example as:

```
(declare-const x (_ BitVec 64))
(declare-const y (_ BitVec 64))
(assert
        (forall ((x (_ BitVec 64)) (y (_ BitVec 64)))
                (=
                        (bvsub
                                (bvadd x y)
                                (bvshl (bvand x y) (_ bv1 64))
                        )
                        (bvxor x y)
                )
        )
)
(check-sat)
```

It returns “sat”, meaning, the equation is correct for all possible 64-bit `x` and `y` values, like them all were checked. Mathematically speaking: $\forall n \in \mathbb{N}\ (x \oplus y = (x + y - ((x\&y) << 1)))$ [3]

4.1.3 How the expression works

First of all, binary addition can be viewed as binary XORing with carrying (2.3.2). Here is an example: let’s add 2 (10b) and 2 (10b). XORing these two values resulting 0, but there is a carry generated during addition of two second bits. That carry bit is propagated further and settles at the place of the 3rd bit: 100b. 4 (100b) is hence a final result of addition.

If the carry bits are not generated during addition, the addition operation is merely XORing. For example, let’s add 1 (1b) and 2 (10b). $1 + 2$ equals to 3, but $1 \oplus 2$ is also 3.

If the addition is XORing plus carry generation and application, we should eliminate effect of carrying somehow here. The first part of the expression $(x + y)$ is addition, the second $((x\&y) << 1)$ is just calculation of every carry bit which was used during addition. If to subtract carry bits from the result of addition, the only XOR effect is left then.

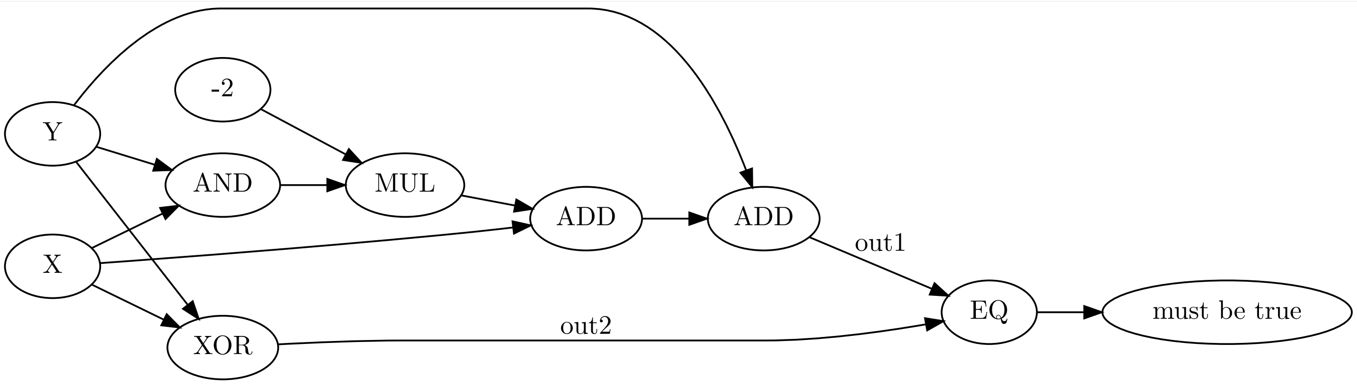
It’s hard to say how Z3 proves this: maybe it just simplifies the equation down to single XOR using simple Boolean algebra rewriting rules?

4.2 Proving bizarre XOR alternative using SAT solver

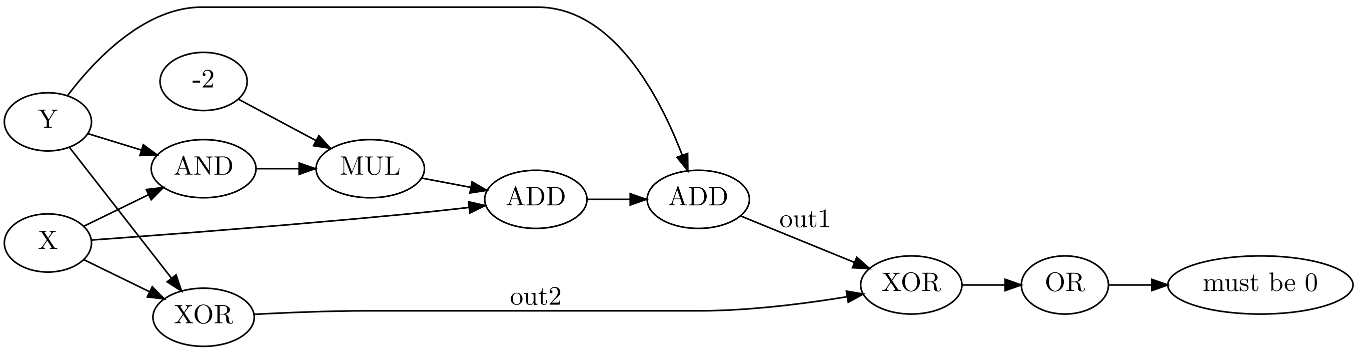
Now let’s try to prove it using SAT.

We would build an electric circuit for $x \oplus y = -2 * (x\&y) + (x + y)$ like that:

[3] $\forall$ means *equation must be true for all possible values*, which are chosen from natural numbers ($\mathbb{N}$).



And now we can implement EQ block using XOR and OR:



So it has two parts: generic XOR block and a block which must be equivalent to XOR. Then we compare its outputs using XOR and OR. If outputs of these parts are always equal to each other for all possible x and y, output of the whole block must be 0.

I do otherwise, I'm trying to find such an input pair, for which output will be 1:

```
def chk1():
    input_bits=8

    s=SAT_lib.SAT_lib(False)

    x,y=s.alloc_BV(input_bits),s.alloc_BV(input_bits)
    step1=s.BV_AND(x,y)
    minus_2=[s.const_true]*(input_bits-1)+[s.const_false]
    product=s.multiplier(step1,minus_2)[input_bits:]
    result1=s.adder(s.adder(product, x)[0], y)[0]

    result2=s.BV_XOR(x,y)

    s.fix(s.OR(s.BV_XOR(result1, result2)), True)

    if s.solve()==False:
        print ("unsat")
        return

    print ("sat")
    print ("x=%x" % SAT_lib.BV_to_number(s.get_BV_from_solution(x)))
    print ("y=%x" % SAT_lib.BV_to_number(s.get_BV_from_solution(y)))
    print ("step1=%x" % SAT_lib.BV_to_number(s.get_BV_from_solution(step1)))
    print ("product=%x" % SAT_lib.BV_to_number(s.get_BV_from_solution(product)))
    print ("result1=%x" % SAT_lib.BV_to_number(s.get_BV_from_solution(result1)))
    print ("result2=%x" % SAT_lib.BV_to_number(s.get_BV_from_solution(result2)))
    print ("minus_2=%x" % SAT_lib.BV_to_number(s.get_BV_from_solution(minus_2)))
```

The full source code: https://smt.st/current_tree/proofs/XOR_SAT/XOR_SAT.py.

SAT solver returns "unsat", meaning, it couldn't find such a pair. In other words, it couldn't find a counterexample. So the circuit always outputs 0, for all possible inputs, meaning, outputs of two parts are always the same.

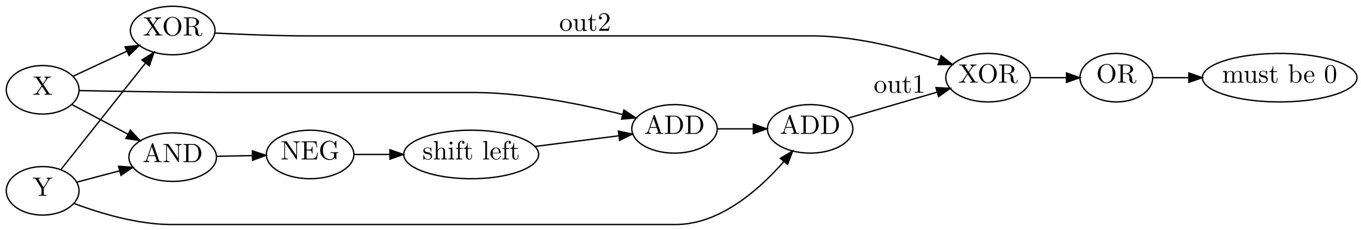
Modify the circuit, and the program will find such a state, and print it.

That circuit also called "miter". According to Google translate, one meaning of the word is:

```
a joint made between two pieces of wood or other material at an angle of 90°, such
    that the line of junction bisects this angle.
```

It's also slow, because multiplier block is used: so we use small 8-bit x's and y's.

But the whole thing can be rewritten: $x \oplus y = x + y - (x\&y) << 1$. And subtraction is addition, but with one negated operand. So, $x \oplus y = (-(x\&y)) << 1 + (x + y)$ or $x \oplus y = (x\&y) * 2 - (x + y)$.



NEG is negation block, in two's complement system. It just inverts all bits and adds 1:

```
    def NEG(self, x):
        # invert all bits
        tmp=self.BV_NOT(x)
        # add 1
        one=self.alloc_BV(len(tmp))
        self.fix_BV(one,n_to_BV(1, len(tmp)))
        return self.adder(tmp, one)[0]
```

Shift by one bit does nothing except rewiring.

That works way faster, and can prove correctness for 64-bit x's and y's, or for even bigger input values:

```
def chk2():
    input_bits=64

    s=SAT_lib.SAT_lib(False)

    x,y=s.alloc_BV(input_bits),s.alloc_BV(input_bits)
    step1=s.BV_AND(x,y)
    step2=s.shift_left_1(s.NEG(step1))

    result1=s.adder(s.adder(step2, x)[0], y)[0]

    result2=s.BV_XOR(x,y)

    s.fix(s.OR(s.BV_XOR(result1, result2)), True)

    if s.solve()==False:
        print ("unsat")
        return

    print ("sat")
    print ("x=%x" % SAT_lib.BV_to_number(s.get_BV_from_solution(x)))
    print ("y=%x" % SAT_lib.BV_to_number(s.get_BV_from_solution(y)))
    print ("step1=%x" % SAT_lib.BV_to_number(s.get_BV_from_solution(step1)))
    print ("step2=%x" % SAT_lib.BV_to_number(s.get_BV_from_solution(step2)))
    print ("result1=%x" % SAT_lib.BV_to_number(s.get_BV_from_solution(result1)))
    print ("result2=%x" % SAT_lib.BV_to_number(s.get_BV_from_solution(result2)))
```

The source code: https://smt.st/current_tree/proofs/XOR_SAT/XOR_SAT.py.

4.3 Dietz's formula

One of the impressive examples of *Aha!* work is finding of Dietz's formula[4], which is the code of computing average number of two numbers without overflow (which is important if you want to find average number of numbers like 0xFFFFFF00 and so on, using 32-bit registers).

Taking this in input:

```
int userfun(int x, int y) {     // To find Dietz's formula for
                                // the floor-average of two
                                // unsigned integers.
   return ((unsigned long long)x + (unsigned long long)y) >> 1;
}
```

...the *Aha!* gives this:

```
Found a 4-operation program:
   and   r1,ry,rx
   xor   r2,ry,rx
   shrs  r3,r2,1
   add   r4,r3,r1
   Expr: (((y ^ x) >>s 1) + (y & x))
```

And it works correctly[5]. But how to prove it?

We will place Dietz's formula on the left side of equation and $x + y/2$ (or $x + y >> 1$) on the right side:

$$\forall n \in 0..2^{64} - 1.(x \& y) + (x \oplus y) >> 1 = x + y >> 1$$

One important thing is that we can't operate on 64-bit values on right side, because result will overflow. So we will zero extend inputs on right side by 1 bit (in other words, we will just 1 zero bit before each value). The result of Dietz's formula will also be extended by 1 bit. Hence, both sides of the equation will have a width of 65 bits:

```
(declare-const x (_ BitVec 64))
(declare-const y (_ BitVec 64))
(assert
        (forall ((x (_ BitVec 64)) (y (_ BitVec 64)))
                (=
                        ((_ zero_extend 1)
                                (bvadd
                                        (bvand x y)
                                        (bvlshr (bvxor x y) (_ bv1 64))
                                )
                        )
                        (bvlshr
                                (bvadd ((_ zero_extend 1) x) ((_ zero_extend 1) y))
                                (_ bv1 65)
                        )
                )
        )
)
(check-sat)
```

Z3 says "sat".

65 bits are enough, because the result of addition of two biggest 64-bit values has width of 65 bits:
`0xFF...FF + 0xFF...FF = 0x1FF...FE`.

As in previous example about XOR equivalent, `(not (= ... ))` and `exists` can also be used here instead of `forall`.

[4] http://aggregate.org/MAGIC/#Average%20of%20Integers

[5] For those who interesting how it works, its mechanics is closely related to the weird XOR alternative we just saw. That's why I placed these two pieces of text one after another.

4.4 XOR swapping algorithm

This is well-known XOR swap algorithm (which don't use additional variable). How it works?

```
1  #!/usr/bin/env python3
2
3  from z3 import *
4
5  init_X, init_Y=BitVecs('init_X init_Y', 32)
6
7  X, Y=init_X, init_Y
8
9  X=X^Y
10 Y=Y^X
11 X=X^Y
12
13 print ("X=", X)
14 print ("Y=", Y)
15
16 s=Solver()
17
18 s.add(init_X^init_Y != X^Y)
19 print (s.check())
```

Now we see a final states of X/Y variables:

```
X= init_X ^ init_Y ^ init_Y ^ init_X ^ init_Y
Y= init_Y ^ init_X ^ init_Y
unsat
```

Z3 gave "unsat", meaning, it can't find any counterexample to the last equation (line 18). Hence, the equation is correct and so is the whole algorithm.

4.4.1 In SMT-LIB form

```
; tested with Z3 and MK85

; prove that XOR swap algorithm is correct.
; https://en.wikipedia.org/wiki/XOR_swap_algorithm

; initial: X1, Y1
;X2 := X1 XOR Y1
;Y3 := Y1 XOR X2
;X4 := X2 XOR Y3
; prove X1=Y3 and Y1=X4 for all

; must be unsat, of course

; needless to say that other SMT solvers may use simplification to prove this, MK85
   can't do it,
; it "proves" on SAT level, by absence of counterexample to the expressions.

(set-logic QF_BV)
(set-info :smt-lib-version 2.0)

(declare-fun x1 () (_ BitVec 32))
(declare-fun y1 () (_ BitVec 32))
(declare-fun x2 () (_ BitVec 32))
(declare-fun y3 () (_ BitVec 32))
(declare-fun x4 () (_ BitVec 32))

(assert (= x2 (bvxor x1 y1)))
```

```
(assert (= y3 (bvxor y1 x2)))
(assert (= x4 (bvxor x2 y3)))

(assert (not (and (= x4 y1) (= y3 x1))))

(check-sat)
```

```
; tested with Z3 and MK85

; prove that XOR swap algorithm (using addition/subtraction) is correct.
; https://en.wikipedia.org/wiki/XOR_swap_algorithm

; initial: X1, Y1
;X2 := X1 ADD Y1
;Y3 := X2 SUB Y1
;X4 := X2 SUB Y3
; prove X1=Y3 and Y1=X4 for all

; must be unsat, of course

; needless to say that other SMT solvers may use simplification to prove this, MK85
    can't do it,
; it "proves" on SAT level, by absence of counterexample to the expressions.

(set-logic QF_BV)
(set-info :smt-lib-version 2.0)

(declare-fun x1 () (_ BitVec 32))
(declare-fun y1 () (_ BitVec 32))
(declare-fun x2 () (_ BitVec 32))
(declare-fun y3 () (_ BitVec 32))
(declare-fun x4 () (_ BitVec 32))

(assert (= x2 (bvadd x1 y1)))
(assert (= y3 (bvsub x2 y1)))
(assert (= x4 (bvsub x2 y3)))

(assert (not (and (= x4 y1) (= y3 x1))))

(check-sat)
```

4.5 Simplifying long and messy expressions using Z3

SMT solvers can optimize whatever you give them at input, and the results of that step can be obtained at the API side.

```
#!/usr/bin/env python3
from z3 import *

x = Int('x')
y = Int('y')
print (simplify(x + y + 2*x + 3))
print (simplify(x < y + x + 2))
print (simplify(And(x + 1 >= 3, x**2 + x**2 + y**2 + 2 >= 5)))
```

Listing 4.1: The result

```
3 + 3*x + y
Not(y <= -2)
And(x >= 2, 2*x**2 + y**2 >= 3)
```

Sometimes, that step is used by all sorts of code deobfuscation tools...

Another interesting tool I used is `BooleanMinimize` function in Wolfram Mathematica [6].

4.6 Simplifying long and messy expressions using Mathematica and Z3

...which can be results of Hex-Rays and/or manual rewriting.

I've added to my RE4B book about Wolfram Mathematica capabilities to minimize expressions [7].

Today I stumbled upon this Hex-Rays output:

```
if ( ( x != 7 || y!=0 ) && (x < 6 || x > 7) )
{
        ...
};
```

Both Mathematica and Z3 (using "simplify" command) can't make it shorter, but I've got that gut feeling there is something redundant.

Let's take a look at the right part of the expression. If x must be less than 6 *OR* greater than 7, then it can hold any value except 6 *AND* 7, right? So I can rewrite this manually:

```
if ( ( x != 7 || y!=0 ) && x != 6 && x != 7) )
{
        ...
};
```

And this is what Mathematica can simplify:

```
In[]:= BooleanMinimize[(x != 7 || y != 0) && (x != 6 && x != 7)]
Out[]:= x != 6 && x != 7
```

y gets reduced.

But am I really right? And why Mathematica and Z3 didn't simplify this at first place?

I can use Z3 to prove that these expressions are equal to each other:

```
#!/usr/bin/env python3

from z3 import *

x=Int('x')
y=Int('y')

s=Solver()

exp1=And(Or(x!=7, y!=0), Or(x<6, x>7))
exp2=And(x!=6, x!=7)

s.add(exp1!=exp2)

print (simplify(exp1)) # no luck, the expression doesn't simplified

print (s.check())
#print (s.model())
```

Z3 can't find counterexample, so it says "unsat", meaning, these expressions are equivalent to each other. So I've rewritten this expression in my code, tests has been passed, etc.

Yes, using both Mathematica and Z3 is overkill, and this is basic boolean algebra, but after ≈ 10 hours of sitting at a front of computer you can make really dumb mistakes, and additional proof that your piece of code is correct is never unwanted.

[6] https://reference.wolfram.com/language/ref/BooleanMinimize.html

[7] https://beginners.re/: Ctrl-f: "Hex-Rays"

4.7 Bit reverse function

This is quite popular function. Unfortunately, such a hackish code is error-prone, an unnoticed typo can easily creep in.

```
#define __constant_bitrev32(x)  \
({                                      \
        u32 ___x = x;                   \
        ___x = (___x >> 16) | (___x << 16);      \
        ___x = ((___x & (u32)0xFF00FF00UL) >> 8) | ((___x & (u32)0x00FF00FFUL) << 8);
              \
        ___x = ((___x & (u32)0xF0F0F0F0UL) >> 4) | ((___x & (u32)0x0F0F0F0FUL) << 4);
              \
        ___x = ((___x & (u32)0xCCCCCCCCUL) >> 2) | ((___x & (u32)0x33333333UL) << 2);
              \
        ___x = ((___x & (u32)0xAAAAAAAAUL) >> 1) | ((___x & (u32)0x55555555UL) << 1);
              \
        ___x;                                                          \
})
```

(https://github.com/torvalds/linux/blob/master/include/linux/bitrev.h)

While you can check all possible 32-bit values in brute-force manner, this is infeasible for 64-bit function(s).

As before, I'm not proving here the function is "correct" in some sense, but I'm proving equivalence of two functions: `bitrev64()` and `bitrev64_unoptimized()`, which uses `bitrev32()`, which in turn uses `bitrev16()`, etc...

```
#!/usr/bin/python3

from z3 import *

# from Henry Warren's "Hacker's Delight", Chapter 7
# Or: https://github.com/torvalds/linux/blob/master/include/linux/bitrev.h

# default right shift in Z3 is arithmetical, so I'm using Z3's LShR() function here,
   which is logical shift right

def bitrev8(x):
    x = LShR(x, 4) | (x << 4)
    x = LShR(x & 0xCC, 2) | ((x & 0x33) << 2)
    x = LShR(x & 0xAA, 1) | ((x & 0x55) << 1)
    return x

# these "unoptimized" versions are constructed like a Russian doll...

def bitrev16_unoptimized(x):
    return (bitrev8(x & 0xff) << 8) | (bitrev8(LShR(x, 8)))

def bitrev32_unoptimized(x):
    return (bitrev16_unoptimized(x & 0xffff) << 16) | (bitrev16_unoptimized(LShR(x,
       16)))

def bitrev32(x):
    x = LShR(x, 16) | (x << 16)
    x = LShR(x & 0xFF00FF00, 8) | ((x & 0x00FF00FF) << 8)
    x = LShR(x & 0xF0F0F0F0, 4) | ((x & 0x0F0F0F0F) << 4)
    x = LShR(x & 0xCCCCCCCC, 2) | ((x & 0x33333333) << 2)
    x = LShR(x & 0xAAAAAAAA, 1) | ((x & 0x55555555) << 1)
    return x

def bitrev64_unoptimized(x):
    # both versions must work:
    return (bitrev32_unoptimized(x & 0xffffffff) << 32) | bitrev32_unoptimized(LShR(x
       , 32))
    #return (bitrev32(x & 0xffffffff) << 32) | bitrev32(LShR(x, 32))

# copypasted from CADO-NFS 2.3.0, http://cado-nfs.gforge.inria.fr/download.html
def bitrev64 (a):
    a = LShR(a, 32) ^ (a << 32)
    m = 0x0000ffff0000ffff
    a = (LShR(a, 16) & m) ^ ((a << 16) & ~m)
    m = 0x00ff00ff00ff00ff
    a = (LShR(a, 8) & m) ^ ((a << 8) & ~m)
    m = 0x0f0f0f0f0f0f0f0f
    a = (LShR(a, 4) & m) ^ ((a << 4) & ~m)
    m = 0x3333333333333333
    a = (LShR(a, 2) & m) ^ ((a << 2) & ~m)
    m = 0x5555555555555555
    a = (LShR(a, 1) & m) ^ ((a << 1) & ~m)
    return a

s=Solver()

x=BitVec('x', 64)

# tests.
# uncomment any.
# must be "unsat" in each case.

s.add(bitrev64(bitrev64_unoptimized(x))!=x)

# these are involutory functions, i.e., f(f(x))=x
#s.add(bitrev64_unoptimized(bitrev64_unoptimized(x))!=x)
#s.add(bitrev64(bitrev64(x))!=x)

# must be "unsat", no counterexample found
print (s.check())
```

The problem is easy enough to be solved using my toy MK85 bitblaster, with only tiny modifications:

```
#!/usr/bin/python3

from MK85 import *

# MK85 uses logical shift right for Python operator >>, so here is it as is...

def bitrev32(x):
    x = (x >> 16) | (x << 16)
    x = ((x & 0xFF00FF00) >> 8) | ((x & 0x00FF00FF) << 8)
    x = ((x & 0xF0F0F0F0) >> 4) | ((x & 0x0F0F0F0F) << 4)
    x = ((x & 0xCCCCCCCC) >> 2) | ((x & 0x33333333) << 2)
    x = ((x & 0xAAAAAAAA) >> 1) | ((x & 0x55555555) << 1)
    return x

def bitrev64_unoptimized(x):
    return (bitrev32(x & 0xffffffff) << 32) | bitrev32(x >> 32)

s=MK85(verbose=0)

def bitrev64 (a):
    a = (a >> 32) ^ (a << 32)
    m = 0x0000ffff0000ffff
    a = ((a >> 16) & m) ^ ((a << 16) & ~m)
    m = 0x00ff00ff00ff00ff
    a = ((a >> 8) & m) ^ ((a << 8) & ~m)
    m = 0x0f0f0f0f0f0f0f0f
    a = ((a >> 4) & m) ^ ((a << 4) & ~m)
    m = 0x3333333333333333
    a = ((a >> 2) & m) ^ ((a << 2) & ~m)
    m = 0x5555555555555555
    a = ((a >> 1) & m) ^ ((a << 1) & ~m)
    return a

x=s.BitVec('x', 64)
y=s.BitVec('y', 64)

# tests.

# check this:
#s.add(bitrev64_unoptimized(x)!=bitrev64(x))

# or this:
s.add(bitrev64(x)==y)
s.add(bitrev64(y)!=x)

# must be False:
print (s.check())
```

4.8 Proving sorting network correctness

Sorting networks are highly popular in electronics, GPGPU and even in SAT encodings: https://en.wikipedia.org/wiki/Sorting_network.

Especially bitonic sorters, which are also sorting networks: https://en.wikipedia.org/wiki/Bitonic_sorter.

Its popularity is probably related to the fact they can be parallelized easily.

They are relatively easy to construct, but, finding a smallest possible is a challenge.

There is a smallest network (only 25 comparators) for 9-channel sorting network:

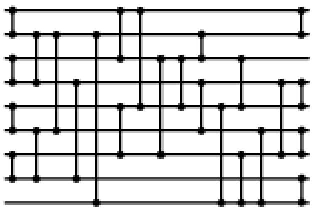


Figure 4.1: Smallest possible

This is combinatorial circuit, each connection is a comparator+swapper, it swaps if one of input values is bigger and passes output to the next level.

I copypasted it from the article: Michael Codish, Lu´ıs Cruz-Filipe, Michael Frank, and Peter Schneider-Kamp – "Twenty-Five Comparators is Optimal when Sorting Nine Inputs (and Twenty-Nine for Ten)".

Another article about it: Ian Parberry – A Computer Assisted Optimal Depth Lower Bound for Nine-Input Sorting Networks.

I don't know (yet) how they proved it, but it's interesting, that it's extremely easy to prove its correctness using Z3 SMT solver. We just construct network out of comparators/swappers and asking Z3 to find counterexample, for which the output of the network will not be sorted. And it can't, meaning, output's state is always sorted, no matter what values are plugged into inputs.

```
#!/usr/bin/env python3
from z3 import *

a, b, c, d, e, f, g, h, i=Ints('a b c d e f g h i')

def Z3_min (a, b):
    return If(a<b, a, b)

def Z3_max (a, b):
    return If(a>b, a, b)

def comparator (a, b):
    return (Z3_min(a, b), Z3_max(a, b))

def line(lst, params):
    rt=lst
    start=0
    while start+1 < len(params):
        try:
            first=params.index("+", start)
        except ValueError:
            # no more "+" in parameter string
            return rt
        second=params.index("+", first+1)
        rt[first], rt[second]=comparator(lst[first], lst[second])
        start=second+1
    # parameter string ended
    return rt

l=[i, h, g, f, e, d, c, b, a]
l=line(l, " ++++++++")
l=line(l, " + + + + ")
l=line(l, "   +   + ")
l=line(l, " +   +   ")
l=line(l, "+      + ")
l=line(l, "  + + + +")
l=line(l, "    +   +")
l=line(l, "  +   +  ")
l=line(l, "    + +  ")
l=line(l, "   + +++ ")
l=line(l, "+   +    ")
l=line(l, "+ + + +  ")
l=line(l, "+  +     ")
l=line(l, "  +  +   ")
l=line(l, "++++++ ++")

# construct expression like And(..., k[2]>=k[1], k[1]>=k[0])
expr=[(l[k+1]>=l[k]) for k in range(len(l)-1)]

# True if everything works correctly:
correct=And(*expr)

s=Solver()

# we want to find inputs for which correct==False:
s.add(Not(correct))
print (s.check()) # must be unsat
```

(The full source code: https://smt.st/current_tree/proofs/sorting_network/test9.py.)

There is also smaller 4-channel network I copypasted from Wikipedia:

```
...

l=line(l, " + +")
l=line(l, "+ + ")
l=line(l, "++++")
l=line(l, " ++ ")
```

```
...
```

(The full source code: https://smt.st/current_tree/proofs/sorting_network/test4.py.)

It also proved to be correct, but it's interesting, what Z3Py expression we've got at each of 4 outputs:

```
If(If(a < c, a, c) < If(b < d, b, d),
   If(a < c, a, c),
   If(b < d, b, d))

If(If(If(a < c, a, c) > If(b < d, b, d),
      If(a < c, a, c),
      If(b < d, b, d)) <
   If(If(a > c, a, c) < If(b > d, b, d),
      If(a > c, a, c),
      If(b > d, b, d)),
   If(If(a < c, a, c) > If(b < d, b, d),
      If(a < c, a, c),
      If(b < d, b, d)),
   If(If(a > c, a, c) < If(b > d, b, d),
      If(a > c, a, c),
      If(b > d, b, d)))

If(If(If(a < c, a, c) > If(b < d, b, d),
      If(a < c, a, c),
      If(b < d, b, d)) >
   If(If(a > c, a, c) < If(b > d, b, d),
      If(a > c, a, c),
      If(b > d, b, d)),
   If(If(a < c, a, c) > If(b < d, b, d),
      If(a < c, a, c),
      If(b < d, b, d)),
   If(If(a > c, a, c) < If(b > d, b, d),
      If(a > c, a, c),
      If(b > d, b, d)))

If(If(a > c, a, c) > If(b > d, b, d),
   If(a > c, a, c),
   If(b > d, b, d))
```

The first and the last are shorter than the 2nd and the 3rd, they are just $min(min(min(a, b), c), d)$ and $max(max(max(a, b), c)$.

Another example in this book related to sorting networks: cracking minesweeper with it (3.10.5).

4.9 ITE example

From [Daniel Kroening and Ofer Strichman — "Decision Procedures, An Algorithmic Point of View", 2ed]:

```
Problem 2.3 (modeling: program equivalence). Show that the two if-then-else
   expressions below are equivalent:

!(a || b) ? h : !(a == b) ? f : g

!(!a || !b) ? g : (!a && !b) ? h : f

You can assume that the variables have only one bit.
```

```
; tested with MK85 and Z3

(set-logic QF_BV)
(set-info :smt-lib-version 2.0)

(declare-fun a () Bool)
(declare-fun b () Bool)
```

```
(declare-fun f () Bool)
(declare-fun g () Bool)
(declare-fun h () Bool)

(declare-fun out1 () Bool)
(declare-fun out2 () Bool)

(assert (=(ite (not (or a b)) h (ite (not (= a b)) f g)) out1))
(assert (=(ite (not (or (not a) (not b))) g (ite (and (not a) (not b)) h f)) out2))

; find counterexample:
(assert (distinct out1 out2))

; must be unsat (no counterexample):
(check-sat)
```

4.10 Branchless abs()

Prove that branchless abs() function from the Henry Warren 2ed, ”2-4 Absolute Value Function” is correct:

```
; tested with Z3 and MK85

(set-logic QF_BV)
(set-info :smt-lib-version 2.0)

(declare-fun x () (_ BitVec 32))

; result1=abs(x), my version:
(declare-fun result1 () (_ BitVec 32))

(assert (= result1 (ite (bvslt x #x00000000) (bvneg x) x)))

; from Henry Warren book.
; y = x>>s 31
; result2=(x xor y) - y
(declare-fun y () (_ BitVec 32))
(declare-fun result2 () (_ BitVec 32))

(assert (= y (bvashr x (_ bv31 32))))
(assert (= result2 (bvsub (bvxor x y) y)))

(assert (distinct result1 result2))

; must be unsat:
(check-sat)
```

4.11 Proving branchless min/max functions are correct

... from https://graphics.stanford.edu/~seander/bithacks.html#IntegerMinOrMax.

Which are, $min(x, y) = y \oplus ((x \oplus y) \wedge (-(x < y)))$

And $max(x, y) = x \oplus ((x \oplus y) \wedge (-(x < y)))$

```
; tested with MK85 and Z3

; unsigned version
(set-logic QF_BV)
(set-info :smt-lib-version 2.0)

(declare-fun x () (_ BitVec 32))
```

```
(declare-fun y () (_ BitVec 32))

(declare-fun min1 () (_ BitVec 32))
(declare-fun max1 () (_ BitVec 32))

; this is our min/max functions, "reference" ones:
(assert (= min1 (ite (bvule x y) x y)))
(assert (= max1 (ite (bvuge x y) x y)))

(declare-fun min2 () (_ BitVec 32))
(declare-fun max2 () (_ BitVec 32))

; functions we will "compare" against:

; y ^ ((x ^ y) & -(x < y)); // min(x, y)
(assert (= min2
        (bvxor
                y
                (bvand
                        (bvxor x y)
                        (bvneg (ite (bvult x y) #x00000001 #x00000000))
                )
        )
))

; x ^ ((x ^ y) & -(x < y)); // max(x, y)
(assert (= max2
        (bvxor
                x
                (bvand
                        (bvxor x y)
                        (bvneg (ite (bvult x y) #x00000001 #x00000000))
                )
        )
))

; find any set of variables for which min1!=min2 or max1!=max2
(assert (or
        (not (= min1 min2))
        (not (= max1 max2))
))

; must be unsat (no counterexample)
(check-sat)
```

```
; tested with MK85 and Z3

; signed version
(set-logic QF_BV)
(set-info :smt-lib-version 2.0)

(declare-fun x () (_ BitVec 32))
(declare-fun y () (_ BitVec 32))

(declare-fun min1 () (_ BitVec 32))
(declare-fun max1 () (_ BitVec 32))

; this is our min/max functions, "reference" ones:
(assert (= min1 (ite (bvsle x y) x y)))
(assert (= max1 (ite (bvsge x y) x y)))
```

```
(declare-fun min2 () (_ BitVec 32))
(declare-fun max2 () (_ BitVec 32))

; functions we will "compare" against:

; y ^ ((x ^ y) & -(x < y)); // min(x, y)
(assert (= min2
        (bvxor
                y
                (bvand
                        (bvxor x y)
                        (bvneg (ite (bvslt x y) #x00000001 #x00000000))
                )
        )
))

; x ^ ((x ^ y) & -(x < y)); // max(x, y)
(assert (= max2
        (bvxor
                x
                (bvand
                        (bvxor x y)
                        (bvneg (ite (bvslt x y) #x00000001 #x00000000))
                )
        )
))

; find any set of variables for which min1!=min2 or max1!=max2
(assert (or
        (not (= min1 min2))
        (not (= max1 max2))
))

; must be unsat (no counterexample)
(check-sat)
```

4.12 Proving “Determine if a word has a zero byte” bit twiddling hack

... which is:

```
#define haszero(v) (((v) - 0x01010101UL) & ~(v) & 0x80808080UL)
```

(https://graphics.stanford.edu/~seander/bithacks.html#ZeroInWord)

The expression returns zero if there are no zero bytes in 32-bit word, or non-zero, if at least one is present. Here we prove that it’s correct for all possible 32-bit words.

```
; checked with Z3 and MK85

(set-logic QF_BV)
(set-info :smt-lib-version 2.0)

(declare-fun v () (_ BitVec 32))
(declare-fun out () (_ BitVec 32))

; (((v) - 0x01010101UL) & ~(v) & 0x80808080UL)
(assert (= out (bvand (bvsub v #x01010101) (bvnot v) #x80808080)))

(declare-fun HasZeroByte () Bool)

(assert (= HasZeroByte
        (or
                (= (bvand v #xff000000) #x00000000)
                (= (bvand v #x00ff0000) #x00000000)
                (= (bvand v #x0000ff00) #x00000000)
                (= (bvand v #x000000ff) #x00000000)
        )
        )
)

; at least one zero byte must be present
(assert HasZeroByte)

; out==0
(assert (= out #x00000000))

; must be unsat (no counterexample)
(check-sat)
```

4.13 Arithmetical shift bit twiddling hack

Prove that `((x+0x80000000) >>u n) - (0x80000000 >>u n)` works like arithmetical shift (`bvashr` function in SMT-LIB or `SAR` x86 instruction).

See: Henry Warren 2ed: ”2-7 Shift Right Signed from Unsigned”.

Also, check if I implemented signed shift right correctly in my MK85:

```
// direction=false for shift left
// direction=true for shift right
// arith=true is for bvashr (only for shifting right)

// for 8-bit left shifter, this is:
// X=ITE(cnt&1, X<<1, X)
// X=ITE((cnt>>1)&1, X<<2, X)
// X=ITE((cnt>>2)&1, X<<4, X)
// i.e., if the bit is set in cnt, shift X by that amount of bits, or do nothing
   otherwise

struct SMT_var* gen_shifter_real (struct ctx* ctx, struct SMT_var* X, struct SMT_var*
    cnt, bool direction, bool arith)
{
        int w=X->width;

        struct SMT_var* in=X;
        struct SMT_var* out;
        struct SMT_var* tmp;

        // bit vector must have width=2^x, i.e., 8, 16, 32, 64, etc
        // FIXME better func name:
        assert (popcount64c (w)==1);

        int bits_in_selector=mylog2(w);

        for (int i=0; i<bits_in_selector; i++)
        {
                if (direction==false)
                        tmp=gen_shift_left(ctx, in, 1<<i);
                else
                        tmp=gen_shift_right(ctx, in, 1<<i, arith ? MSB_of_SMT_var(X)
                            : ctx->var_always_false->SAT_var);

                out=create_internal_variable(ctx, "tmp", TY_BITVEC, w);
                add_Tseitin_ITE_BV (ctx, cnt->SAT_var+i, tmp->SAT_var, in->SAT_var,
                   out->SAT_var, w);

                in=out;
        };
        // if any bit is set in high part of "cnt" variable, result is 0
        // i.e., if a 8-bit bitvector is shifted by cnt>8, give a zero
        struct SMT_var *disable_shifter=create_internal_variable(ctx, "...", TY_BOOL,
            1);
        add_Tseitin_OR_list(ctx, cnt->SAT_var+bits_in_selector, w-bits_in_selector,
           disable_shifter->SAT_var);

        // 0x80 >>s cnt, where cnt>8, must be 0xff! so fill result with MSB(input)
        struct SMT_var *default_val;
        if (arith==false)
                default_val=gen_const(ctx, 0, w);
        else
                default_val=gen_repeat_from_SAT_var(ctx, MSB_of_SMT_var(X), 1, w);

        return gen_ITE(ctx, disable_shifter, default_val, in);
};

struct SMT_var* gen_shifter (struct ctx* ctx, struct SMT_var* X, struct SMT_var* cnt,
    bool direction, bool arith)
{
        int w=X->width;

        // FIXME: better func name:
        if (popcount64c (w)!=1)
        {
                // X is not in 2^n form, so extend it
                // RATIONALE: input must be in $2^n$ form, so the shift count will be
                    $n$
                //printf ("%s() width=%d\n", __FUNCTION__, w);
                int new_w=1<<(mylog2(w)+1);
                //printf ("%s() extending it to width=%d\n", __FUNCTION__, new_w);
                X=gen_zero_extend(ctx, X, new_w-w);
                cnt=gen_zero_extend(ctx, cnt, new_w-w);
        }

        struct SMT_var* rt=gen_shifter_real(ctx, X, cnt, direction, arith);

        if (popcount64c (w)!=1)
        {
                // if X is not in 2^n form
                rt=gen_extract (ctx, rt, 0, w);
        };

        return rt;
};

struct SMT_var* gen_BVASHR (struct ctx* ctx, struct SMT_var* X, struct SMT_var* cnt)
{
        return gen_shifter (ctx, X, cnt, true, true);
};
```

(https://smt.st/MK85)

In other words, we prove equivalence of the expression above and my implementation.

```
; tested with MK85 and Z3

(set-logic QF_BV)
```

```
(set-info :smt-lib-version 2.0)

(declare-fun x () (_ BitVec 32))
(declare-fun n () (_ BitVec 32))
(declare-fun result1 () (_ BitVec 32))
(declare-fun result2 () (_ BitVec 32))

(assert (bvule n (_ bv31 32)))

(assert (= result1 (bvashr x n)))

; ((x+0x80000000) >>u n) - (0x80000000 >>u n)
(assert (= result2
        (bvsub
                (bvlshr (bvadd x #x80000000) n)
                (bvlshr #x80000000 n)
        )
))

(assert (distinct result1 result2))

; must be unsat:
(check-sat)
```

4.14 Proving several floor()/ceiling() properties using Z3

I've found couple problems in [James L. Hein – Discrete Structures, Logic, and Computability]. First is:

```
Find numbers x and y such that floor(x + y) != floor(x) + floor(y) and ceiling(x + y)
    != ceiling(x) + ceiling(y).
```

I can emulate real numbers using fixed point arithmetic over bitvectors. In a 16-bit variable, high 8-bit part will be integer part and low 8-bit part is fractional part. This is also called Q8.8[8].

floor() function is simple — just zero fractional part (low 8 bits). ceiling() function — if *something* is present in fractional part (low 8 bits), increment high 8 bits.

```
from z3 import *

# Find numbers x and y such that floor(x + y) != floor(x) + floor(y) and
# ceiling(x + y) != ceiling(x) + ceiling(y).

def floor(x):
    return x&0xff00

def ceiling(x):
    return If((x&0xff)!=0, (x&0xff00)+0x100, x)

s=Solver()

x,y = BitVecs('x y', 16)

s.add(floor(x+y) != floor(x) + floor(y))
s.add(ceiling(x+y) != ceiling(x) + ceiling(y))

print s.check()
m=s.model()

print "x=0x%04x or %f" % (m[x].as_long(), float(m[x].as_long())/0x100)
print "y=0x%04x or %f" % (m[y].as_long(), float(m[y].as_long())/0x100)
```

[8] https://en.wikipedia.org/wiki/Fixed-point_arithmetic

Listing 4.2: Output

```
sat
x=0x03e0 or 3.875000
y=0x3f20 or 63.125000
```

Let's check this in Wolfram Mathematica:

Listing 4.3: Output of Wolfram Mathematica

```
In[]:= x = 3.875;

In[]:= y = 63.125;

In[]:= {Floor[x + y], Floor[x] + Floor[y]}
Out[]= {67, 66}

In[]:= {Ceiling[x + y], Ceiling[x] + Ceiling[y]}
Out[]= {67, 68}
```

The second problem:

```
from z3 import *

# Find the values of x that satisfy each of the following equations.
# a. ceiling((x - 1)/2) = floor(x/2)
# b. ceiling((x - 1)/3) = floor(x/3)

def floor(x):
    return x&0xff00

def ceiling(x):
    return If((x&0xff)!=0, (x&0xff00)+0x100, x)

s=Solver()

x = BitVec('x', 16)

# 1 in Q8.8 is just 0x0100 or 1<<8
s.add(ceiling((x-0x100)/2) == floor(x/2))
#s.add(ceiling((x-0x100)/3) == floor(x/3))

print s.check()
m=s.model()

print "x=0x%04x or %f" % (m[x].as_long(), float(m[x].as_long())/0x100)
```

Listing 4.4: Output for (a)

```
sat
x=0x4000 or 64.000000
```

Listing 4.5: Output for (b)

```
sat
x=0x0000 or 0.000000
```

The next problem (unsat for all).

IFF[9] is just "==" in Z3, so to find counterexample (if any), we just use "!=".

```
from z3 import *

"""
```

[9] If and only if: $\iff$

```
Prove each of the following statements about inequalities with the floor and ceiling,
where x is a real number and n is an integer.
a. floor(x) < n iff x < n.
b. n < ceiling(x) iff n < x.
c. n <= floor(x) iff n <= x.
d. floor(x) <= n iff x <= n.
"""

def floor(x):
    return x&0xff00

def ceiling(x):
    return If((x&0xff)!=0, (x&0xff00)+0x100, x)

s=Solver()

x = BitVec('x', 16)
n = BitVec('n', 16)

s.add((n&0xff)==0) # n is always integer, it has no fraction

# prevent integer overflow, x and n must be positive
s.add(x<0x8000)
s.add(n<0x8000)

s.add((floor(x) < n) != (x < n)) # a
#s.add((n < ceiling(x)) != (n < x)) # b
#s.add((n <= floor(x)) != (n <= x)) # c
#s.add((floor(x) <= n) != (x <= n)) # d

# must be unsat for a/b/c/d
print s.check()
```

Problem 4, also unsat for all:

```
from z3 import *

"""
Prove each statement, where n is an integer.
a. ceiling(n/2) = floor((n + 1)/2)
b. floor(n/2) = ceiling((n - 1)/2)
"""

def floor(x):
    return x&0xff00

def ceiling(x):
    return If((x&0xff)!=0, (x&0xff00)+0x100, x)

s=Solver()

n = BitVec('n', 16)

s.add((n&0xff)==0) # n is always integer, it has no fraction

# prevent integer overflow, n is always positive
s.add(n<0x8000)

s.add(ceiling(n/2) != (floor((n+0x100)/2))) # a
#s.add(floor(n/2) != (ceiling((n-0x100)/2))) # b

# must be unsat for a/b
```

```
print s.check()
```

4.15 Proving equivalence of two functions using CBMC and Z3 SMT-solver

... while reading the ”Beautiful Code” book, the 5th chapter (”... Lessons from Designing XML Verifiers”) by Elliotte Rusty Harold, I’ve found this piece of code (it’s like isdigit() for Unicode):

```
bool isXMLDigit(unsigned int c)
{
        if (c >= 0x0030 && c <= 0x0039) return true; // ASCII
        if (c >= 0x0660 && c <= 0x0669) return true; // arabic
        if (c >= 0x06F0 && c <= 0x06F9) return true; // arabic
        if (c >= 0x0966 && c <= 0x096F) return true; // Devanagari

        if (c >= 0x09E6 && c <= 0x09EF) return true;

        if (c >= 0x0A66 && c <= 0x0A6F) return true; // Gurmukhi?
        if (c >= 0x0AE6 && c <= 0x0AEF) return true;

        if (c >= 0x0B66 && c <= 0x0B6F) return true;
        if (c >= 0x0BE7 && c <= 0x0BEF) return true;
        if (c >= 0x0C66 && c <= 0x0C6F) return true;

        if (c >= 0x0CE6 && c <= 0x0CEF) return true;
        if (c >= 0x0D66 && c <= 0x0D6F) return true;
        if (c >= 0x0E50 && c <= 0x0E59) return true;

        if (c >= 0x0ED0 && c <= 0x0ED9) return true;
        if (c >= 0x0F20 && c <= 0x0F29) return true; // Tibetan?

        return false;
}
```

... which was then optimized by the author:

```
bool isXMLDigit_optimized(unsigned int c)
{
        if (c < 0x0030) return false; if (c <= 0x0039) return true;
        if (c < 0x0660) return false; if (c <= 0x0669) return true;
        if (c < 0x06F0) return false; if (c <= 0x06F9) return true;
        if (c < 0x0966) return false; if (c <= 0x096F) return true;

        if (c < 0x09E6) return false; if (c <= 0x09EF) return true;
        if (c < 0x0A66) return false; if (c <= 0x0A6F) return true;
        if (c < 0x0AE6) return false; if (c <= 0x0AEF) return true;

        if (c < 0x0B66) return false; if (c <= 0x0B6F) return true;
        if (c < 0x0BE7) return false; if (c <= 0x0BEF) return true;
        if (c < 0x0C66) return false; if (c <= 0x0C6F) return true;

        if (c < 0x0CE6) return false; if (c <= 0x0CEF) return true;
        if (c < 0x0D66) return false; if (c <= 0x0D6F) return true;
        if (c < 0x0E50) return false; if (c <= 0x0E59) return true;

        if (c < 0x0ED0) return false; if (c <= 0x0ED9) return true;
        if (c < 0x0F20) return false; if (c <= 0x0F29) return true;

        return false;
}
```

You see, the problem with such hackish solution is that they are prone to bugs. A small unnoticed (for a long period of time) typo can ruin everything.

4.15.1 CBMC

I am adding this function:

```
void check(unsigned int c)
{
        assert (isXMLDigit_optimized(c) == isXMLDigit(c));
};
```

And asking CBMC to find such an input so that *assert()* would stop:

```
$ cbmc --trace --function check isXMLdigit.c

...

** Results:
[check.assertion.1] assertion isXMLDigit_optimized(c) == isXMLDigit(c): SUCCESS

** 0 of 1 failed (1 iteration)
VERIFICATION SUCCESSFUL
```

Try to alter any line or constant in any function and ...

```
** Results:
[check.assertion.1] assertion isXMLDigit_optimized(c) == isXMLDigit(c): FAILURE

Trace for check.assertion.1:

State 17 file isXMLdigit_bug.c line 64 thread 0
----------------------------------------------------
  INPUT c: 2534u (00000000 00000000 00001001 11100110)

State 20 file isXMLdigit_bug.c line 64 thread 0
----------------------------------------------------
  c=2534u (00000000 00000000 00001001 11100110)

State 24 file isXMLdigit_bug.c line 66 function check thread 0
----------------------------------------------------
  c=2534u (00000000 00000000 00001001 11100110)

State 42 file isXMLdigit_bug.c line 66 function check thread 0
----------------------------------------------------
  c=2534u (00000000 00000000 00001001 11100110)

Violated property:
  file isXMLdigit_bug.c line 66 function check
  assertion isXMLDigit_optimized(c) == isXMLDigit(c)
  return_value_isXMLDigit_optimized == return_value_isXMLDigit
```

c=2534 is an input leading to crash.

4.15.2 Z3 SMT-solver

But I also wanted to know if I can convert all this into propositional logic form and check equivalence using SMT solver.

I would add two types of boolean variables. ”c”-variables for conditions. ”p”-variables are like ”points”. Each ”point” is true if execution flow reaches this point for the corresponding input.

```
bool isXMLDigit(unsigned int c)
{
/*p1.1*/ if (/*c1*/ c>=0x0030 && c<=0x0039) /*p1.2*/ return true;/*p1.3*/
/*p2.1*/ if (/*c2*/ c>=0x0660 && c<=0x0669) /*p2.2*/ return true;/*p2.3*/

...

/*p15.1*/if (/*c15*/c>=0x0F20 && c<=0x0F29) /*p15.2*/return true;/*p15.3*/
/*p16*/
return false;
}
```

p1.1 is always true (we get there for any input). p1.2 is true if p1.1 is true AND c1 condition is true. p1.3 is true if p1.1 is true AND p1.2 is false (as if no ”return true” has been executed). On the next line, p2.1 is a synonym for p1.3.

The function returns false IFF p16==true. The function returns true IFF (p1.2 OR p2.2 OR ... OR p15.2).

Here is how I model this using SMT-LIB 2.0 LISPy language:

```
; /*p1.1*/ if (/*c1*/ c>=0x0030 && c<=0x0039) /*p1.2*/ return true;/*p1.3*/

(assert (= f1_p1_1 true)) ; always
(assert (= f1_c1 (and (bvuge f1_c #x0030) (bvule f1_c #x0039))))
(assert (= f1_p1_2 (and f1_p1_1 f1_c1)))
(assert (= f1_p1_3 (and f1_p1_1 (not f1_p1_2))))

; /*p2.1*/ if (/*c2*/ c>=0x0660 && c<=0x0669) /*p2.2*/ return true;/*p2.3*/

(assert (= f1_p2_1 f1_p1_3))
(assert (= f1_c2 (and (bvuge f1_c #x0660) (bvule f1_c #x0669))))
(assert (= f1_p2_2 (and f1_p2_1 f1_c2)))
(assert (= f1_p2_3 (and f1_p2_1 (not f1_p2_2))))

...

; f1() returns false IFF f1_p16
(assert (= f1_returns_false f1_p16))

; f1() return true IFF f1_p1_2 OR f1_p2_2 OR ... OR f1_p15_2
(assert (= f1_returns_true (or f1_p1_2 f1_p2_2 f1_p3_2 f1_p4_2 f1_p5_2 f1_p6_2
    f1_p7_2 f1_p8_2 f1_p9_2
f1_p10_2 f1_p11_2 f1_p12_2 f1_p13_2 f1_p14_2 f1_p15_2)))
```

Yes, Z3Py could be used, but I chose not to *contaminate* it by Python syntax yet, to get a clearer representation of propositional logic equation.

The optimized version of the function is a bit more complex:

```
bool isXMLDigit_optimized(unsigned int c)
{
/*p1.1*/ if (/*c1.1*/ c<0x0030) /*p1.2*/ return false; /*p1.3*/ if (/*c1.2*/ c<=0
    x0039) /*p1.4*/ return true;/*p1.5*/
/*p2.1*/ if (/*c2.1*/ c<0x0660) /*p2.2*/ return false; /*p2.3*/ if (/*c2.2*/ c<=0
    x0669) /*p2.4*/ return true;/*p2.5*/

...

/*p15.1*/if (/*c15.1*/c<0x0F20) /*p15.2*/return false; /*p15.3*/if (/*c15.2*/c<=0
    x0F29) /*p15.4*/return true;/*p15.5*/
/*p16*/
return false;
}
```

How I model it in SMT:

```
; /*p1.1*/ if (/*c1.1*/ c<0x0030) /*p1.2*/ return false; /*p1.3*/ if (/*c1.2*/ c<=0
    x0039) /*p1.4*/ return true;/*p1.5*/

(assert (= f2_p1_1 true)) ; always
```

```
(assert (= f2_c1_1 (bvult f2_c #x0030)))
(assert (= f2_p1_2 (and f2_p1_1 f2_c1_1)))
(assert (= f2_p1_3 (and f2_p1_1 (not f2_p1_2))))
(assert (= f2_c1_2 (bvule f2_c #x0039)))
(assert (= f2_p1_4 (and f2_p1_3 f2_c1_2)))
(assert (= f2_p1_5 (and f2_p1_1 (not f2_p1_4))))

; /*p2.1*/ if (/*c2.1*/ c<0x0660) /*p2.2*/ return false; /*p2.3*/ if (/*c2.2*/ c<=0
   x0669) /*p2.4*/ return true;/*p2.5*/

(assert (= f2_p2_1 f2_p1_5))
(assert (= f2_c2_1 (bvult f2_c #x0660)))
(assert (= f2_p2_2 (and f2_p2_1 f2_c2_1)))
(assert (= f2_p2_3 (and f2_p2_1 (not f2_p2_2))))
(assert (= f2_c2_2 (bvule f2_c #x0669)))
(assert (= f2_p2_4 (and f2_p2_3 f2_c2_2)))
(assert (= f2_p2_5 (and f2_p2_1 (not f2_p2_4))))
```

And at the very end, I'm asking for such an input for both function, for which their outputs would differ:

```
(assert (= f1_c f2_c))
(assert (or
        (not (= f1_returns_false f2_returns_false))
        (not (= f1_returns_true f2_returns_true))))
```

No, Z3, CVC4 and Boolector can't find such an input, giving `unsat`. The problem is small enough to be tackled by my toy-level MK85 bitblaster. Alter any constant, and SMT solver would find such an input.

Perhaps, this is what CBMC doing internally, if I understand all the things correctly.

4.15.3 The files

https://smt.st/current_tree/proofs/isXML/files

4.15.4 But bruteforce?

You see, you can enumerate all 16-bit inputs effortlessly. Yes, but this is a simple example itself.

Chapter 5

Verification

5.1 Integer overflow

This is a classic bug:

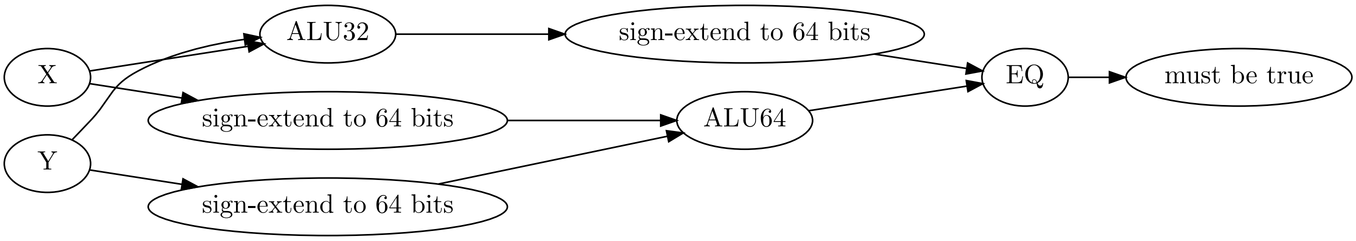
```
void allocate_data_for_some_chunks(int num)
{
#define MAX_CHUNKS 10

        if (num>MAX_CHUNKS)
                // throw error

        void* chunks=malloc(num*sizeof(CHUNK));

        ...
};
```

Seems innocent? However, if a (remote) attacker can put a negative value into *num*, no exception is to be thrown, and `malloc()` will crash on too big input value, because `malloc()` takes unsigned `size_t` on input. *unsigned int* should be used instead of *int* for *num*, but many programmers use *int* as a generic type for everything.

5.1.1 Signed addition

First, let's start with addition. $a + b$ also seems innocent, but it is producing incorrect result if a sum doesn't fit into 32/64-bit register.

This is what we will do: evaluate an expression on two ALUs: 32-bit one and 64-bit one:



In other words, you want your expression to be evaluated on both ALUs correctly, for all possible inputs, right? Like if the result of 32-bit ALU is always fit into 32-bit register.

And now we can ask Z3 SMT solver to find such an a/b inputs, for which the final comparison will fail.

Needless to say, the default operations (+, -, comparisons, etc) in Z3's Python API are signed, you can see this here[1].

Also, we can find the lower bound, or minimal possible inputs, using `minimize()`:

```
#!/usr/bin/env python3
from z3 import *

```

[1] https://github.com/Z3Prover/z3/blob/master/src/api/python/z3/z3.py

```
def func(a,b):
    return a+b

a32, b32, out32 = BitVecs('a32 b32 out32', 32)
out32_extended = BitVec('out32_extended', 64)
a64, b64, out64 = BitVecs('a64 b64 out64', 64)

s=Optimize()

s.add(out32==func(a32, b32))
s.add(out64==func(a64, b64))

s.add(a64==SignExt(32, a32))
s.add(b64==SignExt(32, b32))
s.add(out32_extended==SignExt(32, out32))
s.add(out64!=out32_extended)

s.minimize(a32)
s.minimize(b32)

if s.check()==unsat:
    print ("unsat: everything is OK")
    exit(0)
m=s.model()

# from https://stackoverflow.com/questions/1375897/how-to-get-the-signed-integer-value-of-a-long-in-python
def toSigned32(n):
    n = n & 0xffffffff
    return n | (-(n & 0x80000000))

def toSigned64(n):
    n = n & 0xffffffffffffffff
    return n | (-(n & 0x8000000000000000))

print ("a32=0x%x or %d" % (m[a32].as_long(), toSigned32(m[a32].as_long())))
print ("b32=0x%x or %d" % (m[b32].as_long(), toSigned32(m[b32].as_long())))
print ("out32=0x%x or %d" % (m[out32].as_long(), toSigned32(m[out32].as_long())))

print ("out32_extended=0x%x or %d" % (m[out32_extended].as_long(), toSigned64(m[out32_extended].as_long())))

print ("a64=0x%x or %d" % (m[a64].as_long(), toSigned64(m[a64].as_long())))
print ("b64=0x%x or %d" % (m[b64].as_long(), toSigned64(m[b64].as_long())))
print ("out64=0x%x or %d" % (m[out64].as_long(), toSigned64(m[out64].as_long())))
```

```
a32=0x1 or 1
b32=0x7fffffff or 2147483647
out32=0x80000000 or -2147483648
out32_extended=0xffffffff80000000 or -2147483648
a64=0x1 or 1
b64=0x7fffffff or 2147483647
out64=0x80000000 or 2147483648
```

Right, `1+0x7fffffff = 0x80000000`. But the `0x80000000` value is negative already, because MSB[2] is 1. However, add this on 64-bit ALU and the result will fit in 64-bit register.

How would we fix this problem? We can devise a special function with *wrapped* addition:

```
/* Returns: a + b */
```

[2] Most Significant Bit

```
/* Effects: aborts if a + b overflows */

COMPILER_RT_ABI si_int
__addvsi3(si_int a, si_int b)
{
    si_int s = (su_int) a + (su_int) b;
    if (b >= 0)
    {
        if (s < a)
            compilerrt_abort();
    }
    else
    {
        if (s >= a)
            compilerrt_abort();
    }
    return s;
}
```

(https://github.com/llvm-mirror/compiler-rt/blob/master/lib/builtins/addvsi3.c)

Now I can simulate this function using Z3Py. I'm telling it: "find a solution, where this expression will be false":

```
s.add(Not(If(b32>=0, a32+b32<a32, a32+b32>a32)))
```

And it gives *unsat*, meaning, there is no counterexample, so the expression can be evaluated safely on both ALUs.

But is there a bug in my statement? Let's check. Find inputs for which this piece of LLVM code will call `compilerrt_abort()`:

```
s.add(If(b32>=0, a32+b32<a32, a32+b32>a32))
```

```
a32=0x1 or 1
b32=0x7fffffff or 2147483647
out32=0x80000000 or -2147483648
out32_extended=0xffffffff80000000 or -2147483648
a64=0x1 or 1
b64=0x7fffffff or 2147483647
out64=0x80000000 or 2147483648
```

Safe implementations of other operations: https://wiki.sei.cmu.edu/confluence/display/java/NUM00-J.+Detect+or+prevent+integer+overflow. A popular library: https://github.com/dcleblanc/SafeInt.

5.1.2 Arithmetic mean

Another classic bug. This is the famous bug in binary search algorithms [3]. The bug itself not in binary search algorithm, but in calculating arithmetic mean:

```
def func(a,b):
    return (a+b)/2
```

```
a32=0x1 or 1
b32=0x7fffffff or 2147483647
out32=0xc0000000 or -1073741824
out32_extended=0xffffffffc0000000 or -1073741824
a64=0x1 or 1
b64=0x7fffffff or 2147483647
out64=0x40000000 or 1073741824
```

We can fix this function using a seemingly esoteric Dietz formula, used to do the same, but without integer overflow:

```
def func(a,b):
    return ((a^b)>>1) + (a&b)
```

[3] https://ai.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html, https://thebittheories.com/the-curious-case-of-binary-search-the-famous-bug-that-remained-undetected-for-20-years-973e89fc212, also, see the "Beautiful Code" book (2007), Chapter 7, *That Pesky Binary Search*.

(Its internal workings is described here: 4.3)

Z3 gives *unsat* for this function, because it can't find counterexample.

5.1.3 Allocate memory for some chunks

Let's return to the `allocate_data_for_some_chunks()` function at the beginning of this section.

```
#!/usr/bin/env python3
from z3 import *

def func(a):
    return a*1024

a32, out32 = BitVecs('a32 out32', 32)
out32_extended = BitVec('out32_extended', 64)
a64, out64 = BitVecs('a64 out64', 64)

#s=Solver()
s=Optimize()

s.add(out32==func(a32))
s.add(out64==func(a64))

s.add(a64==SignExt(32, a32))
s.add(out32_extended==SignExt(32, out32))
s.add(out64!=out32_extended)

s.minimize(a32)

if s.check()==unsat:
    print ("unsat: everything is OK")
    exit(0)
m=s.model()

# from https://stackoverflow.com/questions/1375897/how-to-get-the-signed-integer-
    value-of-a-long-in-python
def toSigned32(n):
    n = n & 0xffffffff
    return n | (-(n & 0x80000000))

def toSigned64(n):
    n = n & 0xffffffffffffffff
    return n | (-(n & 0x8000000000000000))

print ("a32=0x%x or %d" % (m[a32].as_long(), toSigned32(m[a32].as_long())))
print ("out32=0x%x or %d" % (m[out32].as_long(), toSigned32(m[out32].as_long())))

print ("out32_extended=0x%x or %d" % (m[out32_extended].as_long(), toSigned64(m[
    out32_extended].as_long())))

print ("a64=0x%x or %d" % (m[a64].as_long(), toSigned64(m[a64].as_long())))
print ("out64=0x%x or %d" % (m[out64].as_long(), toSigned64(m[out64].as_long())))
```

For which *a* values will fail the `a*1024` expression? This is a smallest *a* input:

```
a32=0x200000 or 2097152
out32=0x80000000 or -2147483648
out32_extended=0xffffffff80000000 or -2147483648
a64=0x200000 or 2097152
out64=0x80000000 or 2147483648
```

OK, let's pretend we inserted a `assert (a<100)` before `malloc()` call:

```
s.add(a32<100)
```

```
a32=0x80000000 or -2147483648
out32=0x0 or 0
out32_extended=0x0 or 0
a64=0xffffffff80000000 or -2147483648
out64=0xfffffe0000000000 or -2199023255552
```

Still, an attacker can pass a negative $a = -2147483648$, and `malloc()` will fail.

Let's pretend, we added a `assert (a>0)` before calling `malloc()`:

```
s.add(a32>0)
```

Now Z3 can't find any counterexample.

Some people say you should use functions like `reallocarray()` to be protected from integer overflows: http://lteo.net/blog/2014/10/28/reallocarray-in-openbsd-integer-overflow-detection-for-free/.

5.1.4 abs()

Also seemingly innocent function:

```
def func(a):
    return If(a<0, -a, a)
```

```
a32=0x80000000 or -2147483648
out32=0x80000000 or -2147483648
out32_extended=0xffffffff80000000 or -2147483648
a64=0xffffffff80000000 or -2147483648
out64=0x80000000 or 2147483648
```

This is an artifact of two's complement system. This is `INT_MIN`, and `-INT_MIN == INT_MIN`. It can lead to nasty bugs, and classic one is a naive implementations of `itoa()` or `printf()`.

Take a look at this implementation of `itoa()` function from [Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, 2ed, (1988)]:

```
void itoa(int n, char s[])
{
        int i, sign;
        if ((sign = n) < 0) /* record sign */
                n = -n; /* make n positive */
        i = 0;
        do { /* generate digits in reverse order */
                s[i++] = n % 10 + '0'; /* get next digit */
        } while ((n /= 10) > 0); /* delete it */
        if (sign < 0)
                s[i++] = '-';
        s[i] = '\0';
        strrev(s);
}
```

(The full source code: https://smt.st/current_tree/verif/int_over/itoa_KR.c)

From [Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, 2ed, (1988)]:

> Exercise 3-4. In a two's complement number representation, our version of *itoa* does not handle the largest negative number, that is, the value of n equal to $-(2^{wordsize-1})$. Explain why not. Modify it to print that value correctly, regardless of the machine on which it runs.

The correct answer is: the function cannot process largest negative number (`INT_MIN` or 0x80000000 or -2147483648) correctly.

So, suppose, you print a signed value. And you write something like:

```
if (input<0)
{
        input=-input;
        printf ("-"); // print leading minus
};

// print digits in (positive) input:
...
```

If a INT_MIN value (0x80000000) is passed, minus sign is printed, but the input variable still contain negative value. An additional check for INT_MIN is to be added to fix this.

This is also called *undefined behaviour* in C/C++. The problem is that C language itself is old enough to be a witness of *old iron* – computers which could represent signed numbers in other ways than two's complement representation: ttps://en.wikipedia.org/wiki/Signed_number_representations.

For this reason, C standard doesn't guarantee that −1 will be 0xffffffff (all bits set) on 32-bit registers, because the standard can't guarantee you will run on a hardware with two's complement representation of signed numbers.

However, almost all hardware you can currently use and buy uses two's complement.

More about the abs() problem:

```
This can become a security issue. I have seen one instance in the vasprintf
   implementation of libiberty,
which is part of gcc, binutils and some other GNU software. vasprintf walks over the
   format string and
tries to estimate how much space it will need to hold the formatted result string. In
    format strings,
there is a construct %.*s or %*s, which means that the actual value should be taken
   from the stack.
The libiberty code did it like this:

          if (*p == '*')
            {
              ++p;
              total_width += abs (va_arg (ap, int));
            }

This is actually two issues in one. The first issue is that total_width can overflow.
    The second issue
is the one that is interesting in this context: abs can return a negative number,
   causing the code
to allocate not enough space and thus cause a buffer overflow.
```

(http://www.fefe.de/intof.html)

5.1.5 Games

A lot of video games are prone to integer overflow. Which are exploited actively by gamers. As of NetHack: https://nethackwiki.com/wiki/Integer_overflow, https://nethackwiki.com/wiki/Negative_gold.

5.1.6 Year 2038

And of course, the number of seconds since the year 1970 wouldn't fit in 32-bit variable after 2038.

5.1.7 Summary

What we did here, is we checked, if a result of an expression can fit in 32-bit register. Probably, you can use a narrower second ALU, than a 64-bit one.

5.1.8 Further work

If you want to catch overflows on unsigned variables, use unsigned Z3 operations instead of signed, and do zero extend instead of sign extend.

5.1.9 Some discussion

https://news.ycombinator.com/item?id=18521769

5.1.10 Further reading

- Understanding Integer Overflow in C/C++.
- Modular Bug-finding for Integer Overflows in the Large: Sound, Efficient, Bit-precise Static Analysis.
- blexim – Basic Integer Overflows[4]
- C32SAT.

5.2 Formal verification of population count functions using CBMC

This time we'll prove equivalence of several 64-bit functions, a case when bruteforce isn't feasible.

The population count function is the function that counts number of 1 bits in the input word, AKA Hamming weight.

Many of them I've copypasted from the Wikipedia article[5] about this function and from the chessprogramming.org website[6].

See also John Regehr's blog post[7] about it.

These hackish functions are very arcane, hard to understand and therefore are highly prone to (unnoticed) typos.

I run CBMC and SMT solvers on my ancient Intel Core 2 Duo T9600 clocked at 2.13GHz.

5.2.1 CBMC

So far, CBMC can prove this easily:

```
// time cbmc --trace --function check 1.c

int popcount64a(uint64_t x)
{
        x = (x & m1 ) + ((x >>  1) & m1 );
        x = (x & m2 ) + ((x >>  2) & m2 );
        x = (x & m4 ) + ((x >>  4) & m4 );
        x = (x & m8 ) + ((x >>  8) & m8 );
        x = (x & m16) + ((x >> 16) & m16);
        x = (x & m32) + ((x >> 32) & m32);
        return x;
}

int popcount64b(uint64_t x)
{
        x -= (x >> 1) & m1;
        x = (x & m2) + ((x >> 2) & m2);
        x = (x + (x >> 4)) & m4;
        x += x >>  8;
        x += x >> 16;
        x += x >> 32;
        return x & 0x7f;
}

int popcount64c(uint64_t x)
{
        x -= (x >> 1) & m1;
        x = (x & m2) + ((x >> 2) & m2);
        x = (x + (x >> 4)) & m4;
        return (x * h01) >> 56;
```

[4] http://phrack.org/issues/60/10.html

[5] https://en.wikipedia.org/wiki/Hamming_weight

[6] https://www.chessprogramming.org/Population_Count

[7] https://blog.regehr.org/archives/1667

```
}

int popcount64_naive(uint64_t x)
{
        uint64_t rt=0, i;
        for (i=0; i<64; i++)
                rt=rt+((x>>i)&1);
        return rt;
}

void check(uint64_t c)
{
        assert (popcount64a(c)==popcount64b(c));
        // etc
};
```

These functions maybe quite naive, but I've been interested how CBMC handles arrays:

```
int popcount64_table(uint64_t x)
{
        uint64_t tbl[16]={0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4};

        uint64_t rt=0;

        rt=rt+tbl[(x>>(0*4))&0xf];
        rt=rt+tbl[(x>>(1*4))&0xf];
        ...
        rt=rt+tbl[(x>>(14*4))&0xf];
        rt=rt+tbl[(x>>(15*4))&0xf];

        return rt;
}

int popcount64_table2(uint64_t x)
{
        uint64_t tbl[256]={
0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, 1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4,
   3, 4, 4, 5,
1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, 2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5,
   4, 5, 5, 6,
...
2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, 3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6,
   5, 6, 6, 7,
3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7, 4, 5, 5, 6, 5, 6, 6, 7, 5, 6, 6, 7,
   6, 7, 7, 8};

        uint64_t rt=0;

        rt=rt+tbl[(x>>(0*8))&0xff];
        rt=rt+tbl[(x>>(1*8))&0xff];
        ...
        rt=rt+tbl[(x>>(6*8))&0xff];
        rt=rt+tbl[(x>>(7*8))&0xff];

        return rt;
}
```

Things gets a bit harder with a function copypasted from a well-known HAKMEM. It takes 160 seconds to get job done, despite the somewhat hard (for SAT/SMT solvers) division/remainder function with the odd divisor (63):

```
int hakmem169_32(uint32_t x)
{
   x = x  - ((x >> 1)  & 033333333333)
```

```
          - ((x >> 2)  & 011111111111);
   x = (x +  (x >> 3)) & 030707070707 ;
   return x % 63; /* casting out 63 */
}

int hakmem169_64(uint64_t x)
{
        return hakmem169_32(x>>32) + hakmem169_32(x&0xffffffff);
};
```

Things gets harder on another arcane function attributed to B.Kernighan:

```
int popcount64d(uint64_t x)
{
        int count;
        for (count=0; x; count++)
                x &= x - 1; // x = x & (x-1)
        return count;
}
```

CBMC stuck in seemingly infinite ”unwinding loop”:

```
CBMC version 5.10 (cbmc-5.10) 64-bit x86_64 linux
Parsing 1.c
Converting
Type-checking 1
Generating GOTO Program
Adding CPROVER library (x86_64)
Removal of function pointers and virtual functions
Generic Property Instrumentation
Running with 8 object bits, 56 offset bits (default)
Starting Bounded Model Checking
Unwinding loop popcount64d.0 iteration 1 file 1.c line 50 function popcount64d thread
    0
Unwinding loop popcount64d.0 iteration 2 file 1.c line 50 function popcount64d thread
    0
Unwinding loop popcount64d.0 iteration 3 file 1.c line 50 function popcount64d thread
    0
Unwinding loop popcount64d.0 iteration 4 file 1.c line 50 function popcount64d thread
    0
Unwinding loop popcount64d.0 iteration 5 file 1.c line 50 function popcount64d thread
    0
Unwinding loop popcount64d.0 iteration 6 file 1.c line 50 function popcount64d thread
    0
Unwinding loop popcount64d.0 iteration 7 file 1.c line 50 function popcount64d thread
    0
Unwinding loop popcount64d.0 iteration 8 file 1.c line 50 function popcount64d thread
    0
Unwinding loop popcount64d.0 iteration 9 file 1.c line 50 function popcount64d thread
    0
Unwinding loop popcount64d.0 iteration 10 file 1.c line 50 function popcount64d
   thread 0

...

Unwinding loop popcount64d.0 iteration 1584 file 1.c line 50 function popcount64d
   thread 0
Unwinding loop popcount64d.0 iteration 1585 file 1.c line 50 function popcount64d
   thread 0
Unwinding loop popcount64d.0 iteration 1586 file 1.c line 50 function popcount64d
   thread 0
Unwinding loop popcount64d.0 iteration 1587 file 1.c line 50 function popcount64d
   thread 0
```

```
Unwinding loop popcount64d.0 iteration 1588 file 1.c line 50 function popcount64d
   thread 0

...

etc
```

I couldn't wait it for finish, so I've interrupted.

So I tried to help CBMC by manually limiting the upper bound of the loop:

```
int popcount64d_my(uint64_t x)
{
        int count;
        for (count=0; count<64 && x; count++)
                x &= x - 1;
        return count;
}
```

```
CBMC version 5.10 (cbmc-5.10) 64-bit x86_64 linux
Parsing 1.c
Converting
Type-checking 1
Generating GOTO Program
Adding CPROVER library (x86_64)
Removal of function pointers and virtual functions
Generic Property Instrumentation
Running with 8 object bits, 56 offset bits (default)
Starting Bounded Model Checking
Unwinding loop popcount64d_my.0 iteration 1 file 1.c line 58 function popcount64d_my
   thread 0
Unwinding loop popcount64d_my.0 iteration 2 file 1.c line 58 function popcount64d_my
   thread 0

...

Unwinding loop popcount64d_my.0 iteration 63 file 1.c line 58 function popcount64d_my
    thread 0
Unwinding loop popcount64d_my.0 iteration 64 file 1.c line 58 function popcount64d_my
    thread 0
size of program expression: 517 steps
simple slicing removed 2 assignments
Generated 1 VCC(s), 1 remaining after simplification
Passing problem to propositional reduction
converting SSA
Running propositional reduction
Post-processing
Solving with MiniSAT 2.2.1 with simplifier
17496 variables, 66127 clauses
```

And it takes 42 minutes on my venerable notebook to verify this function.

I also tried to unroll the loop manually:

```
int popcount64d_unrolled(uint64_t x)
{
        int count=0;
        if (x) {x &= x-1; count++;};if (x) {x &= x-1; count++;};if (x) {x &= x-1;
           count++;};if (x) {x &= x-1; count++;};
        if (x) {x &= x-1; count++;};if (x) {x &= x-1; count++;};if (x) {x &= x-1;
           count++;};if (x) {x &= x-1; count++;};

        ...
        if (x) {x &= x-1; count++;};if (x) {x &= x-1; count++;};if (x) {x &= x-1;
           count++;};if (x) {x &= x-1; count++;};
        if (x) {x &= x-1; count++;};if (x) {x &= x-1; count++;};if (x) {x &= x-1;
           count++;};if (x) {x &= x-1; count++;};

        // be sure x==0 upon the function exit...
        assert(x==0);
        return count;
}
```

Now it takes only 16 minutes. Please note the assert() upon the function exit. Yes, ”x” must be ”zeroed” upon the exit for obvious reasons. However, CBMC would prove that the assert() will never throw.

What if I comment one ”if (x)...” line with 4 if’s?

```
SAT checker: instance is SATISFIABLE
Runtime decision procedure: 73.0928s

** Results:
[popcount64d_unrolled.assertion.1] assertion x==0: FAILURE
[check.assertion.1] assertion popcount64d_unrolled(c)==popcount64a(c): FAILURE

Trace for popcount64d_unrolled.assertion.1:

State 24 file 1.c line 164 thread 0
----------------------------------------------------
  INPUT c: 18446743244780863487ul (11111111 11111111 11111111 00111110 11111111
     11111111 11111111 11111111)

State 27 file 1.c line 164 thread 0
----------------------------------------------------
  c=18446743244780863487ul (11111111 11111111 11111111 00111110 11111111 11111111
     11111111 11111111)

State 31 file 1.c line 166 function check thread 0
----------------------------------------------------
  x=18446743244780863487ul (11111111 11111111 11111111 00111110 11111111 11111111
     11111111 11111111)

...
```

You see – the ”x” variable is almost ”filled” with 1’s. Yes, our modified function (with 4 of if’s removed) will run incorrectly when ”x” has 0-3 zero bits.

Also, SAT solver gave ”SATISFIABLE”, meaning, it found a counterexample. Otherwise, it would say ”UNSATISFIABLE”.

Exporting to SMT-LIB 2.x format

... is possible.

As stated in CBMC’s manual:

> CBMC for SMT-LIB Version 2
>
> We also provide a version of CBMC with prototypical support for generating formulas in the SMT-LIB version 2 format. Use the command line options
>
> –smt2 –outfile dest
>
> to write the verification conditions in SMT-LIB version 2 format into the file dest. The CBMC manual is here.

(https://www.cprover.org/SMT-LIB-LSM/)

```
cbmc --smt2 --outfile dest.smt --trace --function check 1.c
```

You can get an idea what CBMC passes to a SMT solver in each case:

```
; SMT 2
(set-info :source "Generated by CBMC 5.10 (cbmc-5.10)")
(set-option :produce-models true)
(set-logic QF_AUFBV)

; set_to true (equal)
(define-fun |__CPROVER_dead_object#1| () (_ BitVec 64) (_ bv0 64))

; set_to true (equal)
(define-fun |__CPROVER_deallocated#1| () (_ BitVec 64) (_ bv0 64))

; set_to true (equal)
(define-fun |__CPROVER_malloc_is_new_array#1| () Bool false)

; set_to true (equal)
(define-fun |__CPROVER_malloc_object#1| () (_ BitVec 64) (_ bv0 64))

; set_to true (equal)
(define-fun |__CPROVER_malloc_size#1| () (_ BitVec 64) (_ bv0 64))

; set_to true (equal)
(define-fun |__CPROVER_memory_leak#1| () (_ BitVec 64) (_ bv0 64))

; set_to true (equal)
(define-fun |__CPROVER_next_thread_id#1| () (_ BitVec 64) (_ bv0 64))

; set_to true (equal)
(define-fun |__CPROVER_pipe_count#1| () (_ BitVec 32) (_ bv0 32))

; set_to true (equal)
(define-fun |__CPROVER_rounding_mode!0#1| () (_ BitVec 32) (_ bv0 32))

; set_to true (equal)
(define-fun |__CPROVER_thread_id!0#1| () (_ BitVec 64) (_ bv0 64))

; the following is a substitute for lambda i. x
(declare-fun array_of.0 () (Array (_ BitVec 64) (_ BitVec 1)))
; set_to true (equal)
(define-fun |__CPROVER_threads_exited#1| () (Array (_ BitVec 64) (_ BitVec 1))
   array_of.0)

; set_to true (equal)
(define-fun |h01#1| () (_ BitVec 64) (_ bv72340172838076673 64))

; set_to true (equal)
(define-fun |m1#1| () (_ BitVec 64) (_ bv6148914691236517205 64))

; set_to true (equal)
(define-fun |m16#1| () (_ BitVec 64) (_ bv281470681808895 64))

; set_to true (equal)
(define-fun |m2#1| () (_ BitVec 64) (_ bv3689348814741910323 64))

; set_to true (equal)
(define-fun |m32#1| () (_ BitVec 64) (_ bv4294967295 64))

; set_to true (equal)
(define-fun |m4#1| () (_ BitVec 64) (_ bv1085102592571150095 64))
```

```
; set_to true (equal)
(define-fun |m8#1| () (_ BitVec 64) (_ bv71777214294589695 64))

; find_symbols
(declare-fun |nondet_symex::nondet0| () (_ BitVec 64))
; set_to true (equal)
(define-fun |__CPROVER__start::c!0#2| () (_ BitVec 64) |nondet_symex::nondet0|)

; set_to true (equal)
(define-fun |check::c!0@1#1| () (_ BitVec 64) |__CPROVER__start::c!0#2|)

; set_to true (equal)
(define-fun |popcount64a::x!0@1#1| () (_ BitVec 64) |check::c!0@1#1|)

; set_to true (equal)
(define-fun |popcount64a::x!0@1#2| () (_ BitVec 64) (bvadd (bvand (_
   bv6148914691236517205 64) (bvlshr |popcount64a::x!0@1#1| ((_ zero_extend 32) (_
   bv1 32)))) (bvand (_ bv6148914691236517205 64) |popcount64a::x!0@1#1|)))

; set_to true (equal)
(define-fun |popcount64a::x!0@1#3| () (_ BitVec 64) (bvadd (bvand (_
   bv3689348814741910323 64) (bvlshr |popcount64a::x!0@1#2| ((_ zero_extend 32) (_
   bv2 32)))) (bvand (_ bv3689348814741910323 64) |popcount64a::x!0@1#2|)))

; set_to true (equal)
(define-fun |popcount64a::x!0@1#4| () (_ BitVec 64) (bvadd (bvand (_
   bv1085102592571150095 64) (bvlshr |popcount64a::x!0@1#3| ((_ zero_extend 32) (_
   bv4 32)))) (bvand (_ bv1085102592571150095 64) |popcount64a::x!0@1#3|)))

; set_to true (equal)
(define-fun |popcount64a::x!0@1#5| () (_ BitVec 64) (bvadd (bvand (_
   bv71777214294589695 64) (bvlshr |popcount64a::x!0@1#4| ((_ zero_extend 32) (_ bv8
   32)))) (bvand (_ bv71777214294589695 64) |popcount64a::x!0@1#4|)))

; set_to true (equal)
(define-fun |popcount64a::x!0@1#6| () (_ BitVec 64) (bvadd (bvand (_
   bv281470681808895 64) (bvlshr |popcount64a::x!0@1#5| ((_ zero_extend 32) (_ bv16
   32)))) (bvand (_ bv281470681808895 64) |popcount64a::x!0@1#5|)))

; set_to true (equal)
(define-fun |popcount64a::x!0@1#7| () (_ BitVec 64) (bvadd (bvand (_ bv4294967295 64)
    (bvlshr |popcount64a::x!0@1#6| ((_ zero_extend 32) (_ bv32 32)))) (bvand (_
   bv4294967295 64) |popcount64a::x!0@1#6|)))

; set_to true (equal)
(define-fun |popcount64a#return_value!0#1| () (_ BitVec 32) ((_ extract 31 0) |
   popcount64a::x!0@1#7|))

; set_to true (equal)
(define-fun |check::$tmp::return_value_popcount64a!0@1#2| () (_ BitVec 32) |
   popcount64a#return_value!0#1|)

; set_to true (equal)
(define-fun |popcount64b::x!0@1#1| () (_ BitVec 64) |check::c!0@1#1|)

; set_to true (equal)
(define-fun |popcount64b::x!0@1#2| () (_ BitVec 64) (bvadd |popcount64b::x!0@1#1| (
   bvneg (bvand (_ bv6148914691236517205 64) (bvlshr |popcount64b::x!0@1#1| ((_
   zero_extend 32) (_ bv1 32)))))))

; set_to true (equal)
```

```
(define-fun |popcount64b::x!0@1#3| () (_ BitVec 64) (bvadd (bvand (_
   bv3689348814741910323 64) (bvlshr |popcount64b::x!0@1#2| ((_ zero_extend 32) (_
   bv2 32)))) (bvand (_ bv3689348814741910323 64) |popcount64b::x!0@1#2|)))

; set_to true (equal)
(define-fun |popcount64b::x!0@1#4| () (_ BitVec 64) (bvand (bvadd (bvlshr |
   popcount64b::x!0@1#3| ((_ zero_extend 32) (_ bv4 32))) |popcount64b::x!0@1#3|) (_
   bv1085102592571150095 64)))

; set_to true (equal)
(define-fun |popcount64b::x!0@1#5| () (_ BitVec 64) (bvadd (bvlshr |popcount64b::x!0
   @1#4| ((_ zero_extend 32) (_ bv8 32))) |popcount64b::x!0@1#4|))

; set_to true (equal)
(define-fun |popcount64b::x!0@1#6| () (_ BitVec 64) (bvadd (bvlshr |popcount64b::x!0
   @1#5| ((_ zero_extend 32) (_ bv16 32))) |popcount64b::x!0@1#5|))

; set_to true (equal)
(define-fun |popcount64b::x!0@1#7| () (_ BitVec 64) (bvadd (bvlshr |popcount64b::x!0
   @1#6| ((_ zero_extend 32) (_ bv32 32))) |popcount64b::x!0@1#6|))

; set_to true (equal)
(define-fun |popcount64b#return_value!0#1| () (_ BitVec 32) (bvand (_ bv127 32) ((_
   extract 31 0) |popcount64b::x!0@1#7|)))

; set_to true (equal)
(define-fun |check::$tmp::return_value_popcount64b!0@1#2| () (_ BitVec 32) |
   popcount64b#return_value!0#1|)

; find_symbols
(declare-fun |__CPROVER__start::c!0#1| () (_ BitVec 64))
; convert
(define-fun |B0| () Bool (= |__CPROVER__start::c!0#1| |__CPROVER__start::c!0#1|))

; find_symbols
(declare-fun |check::$tmp::return_value_popcount64a!0@1#1| () (_ BitVec 32))
; convert
(define-fun |B1| () Bool (= |check::$tmp::return_value_popcount64a!0@1#1| |check::
   $tmp::return_value_popcount64a!0@1#1|))

; find_symbols
(declare-fun |check::$tmp::return_value_popcount64b!0@1#1| () (_ BitVec 32))
; convert
(define-fun |B2| () Bool (= |check::$tmp::return_value_popcount64b!0@1#1| |check::
   $tmp::return_value_popcount64b!0@1#1|))

; set_to false
(assert (not (= |check::$tmp::return_value_popcount64a!0@1#2| |check::$tmp::
   return_value_popcount64b!0@1#2|)))

; find_symbols
(declare-fun |symex::io::0| () (_ BitVec 64))
; set_to true
(assert (= |__CPROVER__start::c!0#2| |symex::io::0|))

(check-sat)

(get-value (|B0|))
(get-value (|B1|))
(get-value (|B2|))
(get-value (|__CPROVER__start::c!0#1|))
```

```
(get-value (|__CPROVER__start::c!0#2|))
(get-value (|__CPROVER_dead_object#1|))
(get-value (|__CPROVER_deallocated#1|))
(get-value (|__CPROVER_malloc_is_new_array#1|))
(get-value (|__CPROVER_malloc_object#1|))
(get-value (|__CPROVER_malloc_size#1|))
(get-value (|__CPROVER_memory_leak#1|))
(get-value (|__CPROVER_next_thread_id#1|))
(get-value (|__CPROVER_pipe_count#1|))
(get-value (|__CPROVER_rounding_mode!0#1|))
(get-value (|__CPROVER_thread_id!0#1|))
(get-value (|__CPROVER_threads_exited#1|))
(get-value (|check::$tmp::return_value_popcount64a!0@1#1|))
(get-value (|check::$tmp::return_value_popcount64a!0@1#2|))
(get-value (|check::$tmp::return_value_popcount64b!0@1#1|))
(get-value (|check::$tmp::return_value_popcount64b!0@1#2|))
(get-value (|check::c!0@1#1|))
(get-value (|h01#1|))
(get-value (|m1#1|))
(get-value (|m16#1|))
(get-value (|m2#1|))
(get-value (|m32#1|))
(get-value (|m4#1|))
(get-value (|m8#1|))
(get-value (|nondet_symex::nondet0|))
(get-value (|popcount64a#return_value!0#1|))
(get-value (|popcount64a::x!0@1#1|))
(get-value (|popcount64a::x!0@1#2|))
(get-value (|popcount64a::x!0@1#3|))
(get-value (|popcount64a::x!0@1#4|))
(get-value (|popcount64a::x!0@1#5|))
(get-value (|popcount64a::x!0@1#6|))
(get-value (|popcount64a::x!0@1#7|))
(get-value (|popcount64b#return_value!0#1|))
(get-value (|popcount64b::x!0@1#1|))
(get-value (|popcount64b::x!0@1#2|))
(get-value (|popcount64b::x!0@1#3|))
(get-value (|popcount64b::x!0@1#4|))
(get-value (|popcount64b::x!0@1#5|))
(get-value (|popcount64b::x!0@1#6|))
(get-value (|popcount64b::x!0@1#7|))
(get-value (|symex::io::0|))

(exit)
; end of SMT2 file
```

5.2.2 SMT solver

Can we do the same using SMT solver?

Here I am encoding popcount64a() function and also its ”naive” counterpart in SMT-LIB language:

```
(declare-fun c0 () (_ BitVec 64)) (assert (= c0 (_ bv0 64))) ; always 0
(declare-fun c1 () (_ BitVec 64)) (assert (= c1 (_ bv1 64))) ; always 1

(declare-fun m1  () (_ BitVec 64)) (assert (= m1 #x5555555555555555))
(declare-fun m2  () (_ BitVec 64)) (assert (= m2 #x3333333333333333))
(declare-fun m4  () (_ BitVec 64)) (assert (= m4 #x0f0f0f0f0f0f0f0f))
(declare-fun m8  () (_ BitVec 64)) (assert (= m8 #x00ff00ff00ff00ff))
(declare-fun m16 () (_ BitVec 64)) (assert (= m16 #x0000ffff0000ffff))
(declare-fun m32 () (_ BitVec 64)) (assert (= m32 #x00000000ffffffff))
```

```
...

(assert (= a_x1  (bvadd (bvand a_x m1)   (bvand (bvlshr a_x  (_ bv1 64)) m1))))
(assert (= a_x2  (bvadd (bvand a_x1 m2)  (bvand (bvlshr a_x1 (_ bv2 64)) m2))))
(assert (= a_x3  (bvadd (bvand a_x2 m4)  (bvand (bvlshr a_x2 (_ bv4 64)) m4))))
(assert (= a_x4  (bvadd (bvand a_x3 m8)  (bvand (bvlshr a_x3 (_ bv8 64)) m8))))
(assert (= a_x5  (bvadd (bvand a_x4 m16) (bvand (bvlshr a_x4 (_ bv16 64)) m16))))
(assert (= a_out (bvadd (bvand a_x5 m32) (bvand (bvlshr a_x5 (_ bv32 64)) m32))))

; popcount64(): the naive way:
; (x>>0)&1 + (x>>1)&1 + (x>>2)&1 + ...

(assert (= naive_out
(bvadd
       (bvand (bvlshr naive_x (_ bv0 64)) (_ bv1 64))
       (bvand (bvlshr naive_x (_ bv1 64)) (_ bv1 64))

       ...

       (bvand (bvlshr naive_x (_ bv62 64)) (_ bv1 64))
       (bvand (bvlshr naive_x (_ bv63 64)) (_ bv1 64))
)))

...

(assert (= naive_x a_x)) (assert (not (= naive_out a_out)))
```

Kernighan's unrolled function a bit tougher:

```
(declare-fun kern_x0 () (_ BitVec 64))
(declare-fun kern_x1 () (_ BitVec 64))
(assert (= kern_x1 (ite (= kern_x0 c0) kern_x0 (bvand kern_x0 (bvsub kern_x0 c1)))))
(declare-fun kern_x2 () (_ BitVec 64))
(assert (= kern_x2 (ite (= kern_x1 c0) kern_x1 (bvand kern_x1 (bvsub kern_x1 c1)))))

...

(declare-fun kern_x63 () (_ BitVec 64))
(assert (= kern_x63 (ite (= kern_x62 c0) kern_x62 (bvand kern_x62 (bvsub kern_x62 c1)
   ))))
(declare-fun kern_x64 () (_ BitVec 64))
(assert (= kern_x64 (ite (= kern_x63 c0) kern_x63 (bvand kern_x63 (bvsub kern_x63 c1)
   ))))

; be sure x==0 upon the function exit...
;(assert (not (= kern_x64 c0)))

(declare-fun kern_out () (_ BitVec 64))

; add 1 to kern_out each time if kern_xX!=0:
(assert (= kern_out (bvadd
        (ite (= kern_x0 c0) c0 c1) (ite (= kern_x1 c0) c0 c1) (ite (= kern_x2 c0) c0
           c1) (ite (= kern_x3 c0) c0 c1)
        (ite (= kern_x4 c0) c0 c1) (ite (= kern_x5 c0) c0 c1) (ite (= kern_x6 c0) c0
           c1) (ite (= kern_x7 c0) c0 c1)

        ...

        (ite (= kern_x56 c0) c0 c1) (ite (= kern_x57 c0) c0 c1) (ite (= kern_x58 c0)
           c0 c1) (ite (= kern_x59 c0) c0 c1)
        (ite (= kern_x60 c0) c0 c1) (ite (= kern_x61 c0) c0 c1) (ite (= kern_x62 c0)
           c0 c1) (ite (= kern_x63 c0) c0 c1)
)))

...

(assert (= naive_x kern_x0)) (assert (not (= naive_out kern_out)))
```

Again, we can prove here that the last state of "x" would always be zero.

Maybe I encoded the problem in a wrong way, but this time, both CVC4 and Z3 stuck and couldn't solve anything withing 15 minutes timeout. However, Boolector can prove it for 1 minute.

Of course, I could misunderstood something.

SMT arrays

Now what about array encoding?

```
;int popcount64_table(uint64_t x)
;{
;       uint64_t tbl[16]={0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4};
;       uint64_t rt=0;
;       rt=rt+tbl[(x>>(0*4))&0xf];
;       rt=rt+tbl[(x>>(1*4))&0xf];
;...

; fill table of constants:
(declare-const tbl (Array (_ BitVec 64) (_ BitVec 64)))
(assert (= tbl (store tbl (_ bv0 64) (_ bv0 64))))
(assert (= tbl (store tbl (_ bv1 64) (_ bv1 64))))
...
(assert (= tbl (store tbl (_ bv14 64) (_ bv3 64))))
(assert (= tbl (store tbl (_ bv15 64) (_ bv4 64))))

(declare-fun tbl_x () (_ BitVec 64))
(declare-fun tbl_out () (_ BitVec 64))

;       rt=rt+tbl[(x>>(0*4))&0xf];
;       rt=rt+tbl[(x>>(1*4))&0xf];
;       ...
(assert (= tbl_out
(bvadd
       (select tbl (bvand (bvlshr tbl_x (_ bv0 64)) (_ bv15 64)))
       (select tbl (bvand (bvlshr tbl_x (_ bv4 64)) (_ bv15 64)))
       ...
       (select tbl (bvand (bvlshr tbl_x (_ bv56 64)) (_ bv15 64)))
       (select tbl (bvand (bvlshr tbl_x (_ bv60 64)) (_ bv15 64)))
)))
```

Here we use "store" function to populate array. At each line, tbl is reassigned. We then use "select" function to address elements of the array.

Arrays are implemented in SMT solvers using UF, but logically, you can think about them as about chains of ITE's (if-then-else). Or like a switch() construct found in many popular PLs.

Even more, Boolector during model generation ("get-model"), shows arrays as ITE chains:

```
  (define-fun f (
   (f_x0 (_ BitVec 64))) (_ BitVec 64)
    (ite (= f_x0 #b0000000000000000000000000000000000000000000000000000000000000000)
       #b0000000000000000000000000000000000000000000000000000000000000000
    (ite (= f_x0 #b0000000000000000000000000000000000000000000000000000000000000001)
       #b0000000000000000000000000000000000000000000000000000000000000001

...

    (ite (= f_x0 #b0000000000000000000000000000000000000000000000000000000000001111)
       #b0000000000000000000000000000000000000000000000000000000000000100
```

```
      #b0000000000000000000000000000000000000000000000000000000000000000)))))))))))))
         ))))
```

(We see here that the default value is 0, it's like "default" in "switch()" statement.)

As well as Z3:

```
  (define-fun f ((x!0 (_ BitVec 64))) (_ BitVec 64)
    (ite (= x!0 #x0000000000000003) #x0000000000000002
    (ite (= x!0 #x0000000000000004) #x0000000000000001
    ...
    (ite (= x!0 #x0000000000000002) #x0000000000000001
    (ite (= x!0 #x0000000000000009) #x0000000000000002
      #x0000000000000000))))))))))))))))
```

You can think about "select" function as if it simply evaluates internal ITE chain. And "store" function simply prepends new ITE in front of an array (or ITE chain) returning a new array (or chain).

Anyway, this version Boolector can solve for 15 seconds, Z3 for 7 minutes and CVC4 is stuck.

Uninterpreted functions

Since my array is a constant one, I can try to implement it using UF.

Here f() acts like the "select" function. And it's populated like: `assert f(0) is 0, f(1) is 1 ... f(15) is (4)`

```
(declare-fun f ((_ BitVec 64)) (_ BitVec 64))
(assert (= (f (_ bv0 64)) (_ bv0 64)))
(assert (= (f (_ bv1 64)) (_ bv1 64)))
...
(assert (= (f (_ bv14 64)) (_ bv3 64)))
(assert (= (f (_ bv15 64)) (_ bv4 64)))

(declare-fun f_x () (_ BitVec 64))
(declare-fun f_out () (_ BitVec 64))

;       rt=rt+f((x>>(0*4))&0xf);
;       rt=rt+f((x>>(1*4))&0xf);
;       ...
(assert (= f_out
(bvadd
       (f (bvand (bvlshr f_x (_ bv0 64)) (_ bv15 64)))
       (f (bvand (bvlshr f_x (_ bv4 64)) (_ bv15 64)))
       ...
       (f (bvand (bvlshr f_x (_ bv56 64)) (_ bv15 64)))
       (f (bvand (bvlshr f_x (_ bv60 64)) (_ bv15 64)))
)))
```

Boolector and Z3 can solve this for 10-15 seconds, CVC4 is stuck.

Moral of the story

SAT/SMT solvers are highly capricious for various types of input. So it's good idea to try several of them. And of course, I probably misunderstood something, don't know yet, what exactly.

5.2.3 Files used

Here: https://smt.st/current_tree/verif/popcount/files.

5.3 Knuth-Morris-Pratt string-searching algorithm

5.3.1 My homebrew algorithms formally verified using CBMC

Searching for the "ok" substring

Imagine you want to find an "ok" substring within a string. You would do:

```
unsigned search_ok_1 (char *s, unsigned len)
{
        if (len<2)
                return len;       // not found
        for (unsigned i=0; i<len-1; i++)
        {
                if (s[i]=='o' && s[i+1]=='k')
                        return i; // found
        };
        return len;               // not found
};
```

You see, there are two (cache-)memory accesses per one character of the input string (at average). Hence, the total number of all (cache-)memory accesses can be len*2 at worst.

Is it possible to reduce that number? Yes. In the following example, we have only one single memory access per character:

```
unsigned search_ok_2 (char *s, unsigned len)
{
        if (len<2)
                return len;           // not found
        bool seen_o=false;
        for (unsigned i=0; i<len; i++)
        {
                char ch=s[i];         // this is single read operation
                if (ch=='o')
                        seen_o=true;
                else if (seen_o && ch=='k')
                        return i-1;   // found
                else
                        seen_o=false; // reset
        };
        return len; // not found
};
```

Searching for the "eel" substring, verification using CBMC

Can we extend that method to any 3-character strings? Let's write a function that searches for the 'eel' string [8]

We will use the 'seen' variable, reflecting, how many characters of the sought string we've already seen:

```
unsigned search_eel (char *s, unsigned len)
{
        if (len<3)
                return len;         // not found
        unsigned seen=0;
        for (unsigned i=0; i<len; i++)
        {
                char ch=s[i];       // this is single read operation
                if (seen==0 && ch=='e')
                        seen=1;
                else if (seen==1 && ch=='e')
                        seen=2;
                else if (seen==2 && ch=='l')
                        return i-2; // found
                else
                        seen=0;     // reset
        }
        return len;                 // not found
};
```

[8] For non-English speakers: this is indeed a word. Some kind of fish.

But this implementation has a bug. To find it, I'll use CBMC, the excellent tool, that can verify if a C function is equivalent to another function.

This is like Test-driven development (TDD).

I'll add a 'reference' (though, slow) function, but always correct:

```
unsigned search_eel_brute (char *s, unsigned len)
{
        if (len<3)
                return len; // not found
        for (unsigned i=0; i<len-2; i++)
        {
                if (s[i]=='e' && s[i+1]=='e' && s[i+2]=='l')
                        return i;
        };
        return len; // not found
};

...

void check()
{
        unsigned len=LEN;
        char s[len];
        __CPROVER_assert (search_eel_brute(s, len)==search_eel(s, len), "assert");
};
```

```
% cbmc --trace --function check -DLEN=4 kmp_eel2.c
...

** Results:
[check.assertion.1] assert: FAILURE

Trace for check.assertion.1:

...

State 21 file kmp_eel2.c line 41 function check thread 0
----------------------------------------------------
  s={ 'e', 'e', 'e', 'l' } ({ 01100101, 01100101, 01100101, 01101100 })
...
```

It failed for the string "eeel". After some thinking, we can find a problem. If a third character isn't 'l', but 'e', we are in the middle of a long string of 'e' characters. So if seen==2 and the input character isn't 'l', but 'e', we shouldn't advance the 'seen' variable:

```
...
                if (seen==0 && ch=='e')
                        seen=1;
                else if (seen==1 && ch=='e')
                        seen=2;
                else if (seen==2 && ch=='l')
                        return i-2;             // found
                else if (seen==2 && ch=='e')    // fix
                        seen=2;                 // fix
                else
                        seen=0;                 // reset
...
```

Now the verification is successful up to strings of length 15:

```
#!/bin/bash
for i in $( seq 0 15 ); do
    #echo $i
    cbmc --trace --function check -DLEN=$i kmp_eel2.c
done
```

Searching for the ”cocos” substring, verification using CBMC

The ”cocos” substring is much more problematic. But fixing it with the help of CBMC is a great programming exercise.

```
unsigned search_cocos_brute (char *s, unsigned len)
{
        if (len<5)
                return len;       // not found
        for (unsigned i=0; i<len-4; i++)
        {
                if (s[i]=='c' &&
                                s[i+1]=='o' &&
                                s[i+2]=='c' &&
                                s[i+3]=='o' &&
                                s[i+4]=='s')
                        return i; // found
        };
        return len;               // not found
};

unsigned search_cocos_naive (char *s, unsigned len)
{
        if (len<5)
                return len;          // not found
        unsigned seen=0;
        for (unsigned i=0; i<len; i++)
        {
                char ch=s[i];        // this is single read operation
                if (seen==0 && ch=='c')
                        seen=1;
                else if (seen==1 && ch=='o')
                        seen=2;
                else if (seen==2 && ch=='c')
                        seen=3;
                else if (seen==3 && ch=='o')
                        seen=4;
                else if (seen==4 && ch=='s')
                        return i-4;  // found
                else
                        seen=0;      // reset
        }
        return len;                  // not found
};

void check()
{
        unsigned len=LEN;
        char s[len];
        __CPROVER_assert (search_cocos_brute(s, len)==search_cocos_naive(s, len), "
           assert");
};
```

This is the first bug:

```
% cbmc --trace --function check -DLEN=6 kmp_cocos.c
...
** Results:
[check.assertion.1] assert: FAILURE

Trace for check.assertion.1:
```

```
...

State 21 file kmp_cocos.c line 47 function check thread 0
----------------------------------------------------
  s={ 'c', 'c', 'o', 'c', 'o', 's' } ({ 01100011, 01100011, 01101111, 01100011,
     01101111, 01110011 })

...
```

A test failed for the ”ccocos” string.

Here is the fix: if the first ’c’ is repeating, we shouldn’t advance the ’seen’ variable:

```
                if (seen==0 && ch=='c')
                        seen=1;
                else if (seen==1 && ch=='o')
                        seen=2;
                else if (seen==1 && ch!='o')
                {
                                     // we can be here if the input is 'ccocos'
                        if (ch=='c')
                                seen=1;
                        else
                                seen=0;
                }
                else if (seen==2 && ch=='c')
                        seen=3;
                else if (seen==3 && ch=='o')
                        seen=4;
                else if (seen==4 && ch=='s')
                        return i-4;  // found
                else
                        seen=0;      // reset
```

CBMC can verify this function for all 6-character strings, OK. But it can find problematic 7-character string:

```
% cbmc --trace --function check -DLEN=7 kmp_cocos.c
...
** Results:
[check.assertion.1] assert: FAILURE

Trace for check.assertion.1:

...
State 21 file kmp_cocos.c line 79 function check thread 0
----------------------------------------------------
  s={ 'c', 'o', 'c', 'o', 'c', 'o', 's' } ({ 01100011, 01101111, 01100011, 01101111,
     01100011, 01101111, 01110011 })
...
```

That is, ”cococos”.

And this is my fix:

```
...
                else if (seen==4 && ch=='s')
                        return i-4;  // found
                else if (seen==4 && ch!='s')
                {
                        // the input string is 'cocoX' where X is not 's'
                        // (current state of ch='X')
                        // but 'X' could be 'c' if the input string is 'cococos'
                        if (ch=='c')
                        {
                                // if the string is 'cococos',
                                // we can say that we have already seen the 'coc'
                                   part of it:
                                seen=3;
                        }
                        else
                        {
                                // 'X' is not 'c', so reset
                                seen=0;
                        };
                }
                else
                        seen=0;      // reset
...
```

Now all 7-characters strings can be tested without a fail. But not 8-character ones:

```
% cbmc --trace --function check -DLEN=8 kmp_cocos.c
...

** Results:
[check.assertion.1] assert: FAILURE

Trace for check.assertion.1:

...

State 21 file kmp_cocos.c line 96 function check thread 0
----------------------------------------------------
  s={ 'c', 'o', 'c', 'c', 'o', 'c', 'o', 's' } ({ 01100011, 01101111, 01100011,
     01100011, 01101111, 01100011, 01101111, 01110011 })

...
```

It fails with ”coccocos”. We have to add another check for the repeating second ’c’ character.

```
...
                else if (seen==3 && ch=='o')
                        seen=4;
                else if (seen==3 && ch!='o')
                {
                        // if input='coccocos'
                        if (ch=='c')
                                seen=1;
                        else
                                seen=0; // reset
                }
                else if (seen==4 && ch=='s')
...
```

Now CBMC can check it all up to 15-character strings. The whole fixed function is:

```
unsigned search_cocos_fixed (char *s, unsigned len)
{
        if (len<5)
                return len;          // not found
        unsigned seen=0;
        for (unsigned i=0; i<len; i++)
        {
                char ch=s[i];        // this is single read operation
                if (seen==0 && ch=='c')
                        seen=1;
                else if (seen==1 && ch=='o')
```

```
                        seen=2;
                else if (seen==1 && ch!='o')
                {
                                     // we can be here if the input is 'ccocos'
                        if (ch=='c')
                                seen=1;
                        else
                                seen=0;
                }
                else if (seen==2 && ch=='c')
                        seen=3;
                else if (seen==3 && ch=='o')
                        seen=4;
                else if (seen==3 && ch!='o')
                {
                        // if input='coccocos'
                        if (ch=='c')
                                seen=1;
                        else
                                seen=0; // reset
                }
                else if (seen==4 && ch=='s')
                        return i-4;  // found
                else if (seen==4 && ch!='s')
                {
                        // the input string is 'cocoX' where X is not 's'
                        // (current state of ch='X')
                        // but 'X' could be 'c' if the input string is 'cococos'
                        if (ch=='c')
                        {
                                // if the string is 'cococos',
                                // we can say that we have already seen the 'coc'
                                   part of it:
                                seen=3;
                        }
                        else
                        {
                                // 'X' is not 'c', so reset
                                seen=0;
                        };
                }
                else
                        seen=0;      // reset
        }
        return len;                  // not found
};
```

It is capable of searching for the ’cocos’ substring reading each character of the input string only once, and it is formally verified by CBMC.

Code like that is very hard to test (can you execute these functions with all 15-characters input strings?), but thanks to CBMC, we can be sure it’s correct, or at least, equivalent to the simple ’bruteforce’ version. I couldn’t devise a correct version without it. In fact first versions were written in Python. I rewritten it to pure C so that I can verify them using CBMC.

All the files used

https://smt.st/current_tree/verif/KMP/files_1

But what about bruteforce?

Naive way of testing would be feeding all possible 15-character strings, that is: 256^{15} test strings. But you can probably use only characters that are present in a substring: ’c’, ’o’, ’s’ for ’cocos’, plus a character acting as *everything else*, say, '_'. That would result in $4^{15} = 1,073,741,824$ test strings. Yes, it's possible.

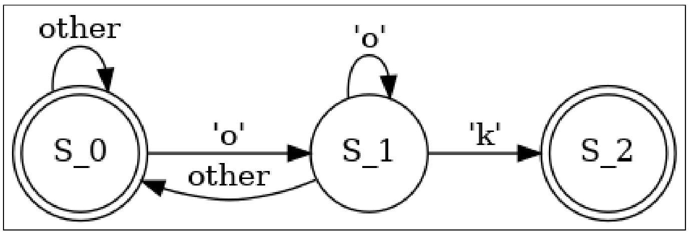
5.3.2 The DFA version

I reworked the Java code by Robert Sedgewick from his excellent book, and rewritten it to Python:

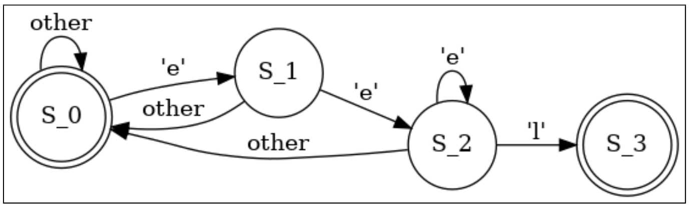
```
#!/usr/bin/env python3

def KMP(pat):
    R=256
    m=len(pat)

    # build DFA from pattern
    # https://stackoverflow.com/questions/2397141/how-to-initialize-a-two-dimensional-array-in-python
    dfa=[[0]*m for r in range(R)]
    dfa[ord(pat[0])][0]=1
    x=0
    for j in range(1, m):
        for c in range(R):
            dfa[c][j] = dfa[c][x]       # Copy mismatch cases.
        dfa[ord(pat[j])][j] = j+1       # Set match case.
        x = dfa[ord(pat[j])][x]         # Update restart state.

    return dfa

# https://stackoverflow.com/questions/3525953/check-if-all-values-of-iterable-are-zero
def all_elements_in_list_are_zeros (l):
    return all(v==0 for v in l)

def export_dfa_to_graphviz(dfa, fname):
    m=len(dfa[0]) # len of pattern
    f=open(fname,"w")
    f.write("digraph finite_state_machine {\n")
    f.write("rankdir=LR;\n")
    f.write("\n")
    f.write("size=\"8,5\"\n")
    f.write(f"node [shape = doublecircle]; S_0 S_{m};\n")
    f.write("node [shape = circle];\n")

    for state in range(m):
        exits=[]
        for R in range(256):
            next=dfa[R][state]
            if next!=0:
                exits.append((R,next))
        for exit in exits:
            next_state=exit[1]
            label="'"+chr(exit[0])+"'"
            s=f"S_{state} -> S_{next_state} [ label = \"{label}\" ];\n"
            f.write (s)
        s=f"S_{state} -> S_0 [ label = \"other\" ];\n"
        f.write (s)
    f.write("}\n")
    f.close()
    print (f"{fname} written")

def search(dfa, txt):
    # simulate operation of DFA on text
    m=len(dfa[0]) # len of pattern
    n=len(txt)
    j=0 # FA state
    i=0
    while i<n and j<m:
        j = dfa[ord(txt[i])][j]
        i=i+1
    if j == m:
        return i - m # found
    return n         # not found
```

I added a code to produce a graph in GraphViz format.

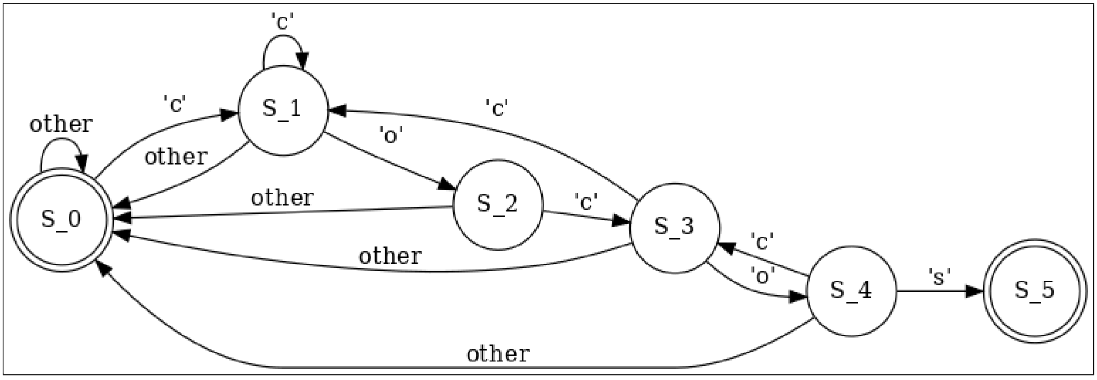
Now what about the 'ok' word?



Another word we already know, 'eel':



The 'cocos':



You see, I can translate these FA[9]s to Python code:

```
#!/usr/bin/env python3

def search_eel(txt):
    # simulate operation of DFA on text
    m=3 # len of pattern
    n=len(txt)
    j=0 # FA state
    i=0 # iterator for txt[]
    while i<n and j<m:
        ch=txt[i]
        if j==0 and ch=='e':
            j=1
        elif j==1 and ch=='e':
            j=2
```

[9] Finite Automaton

```
        elif j==2 and ch=='e':
            j=2
        elif j==2 and ch=='l':
            j=3
        else:
            j=0 # reset
        i=i+1
    if j == m:
        return i - m # found
    return n         # not found
```

Do you see any similarities with the C code from the previous part 5.3.1?

Yes, what we actually did, was FA, and the 'seen' variable was used as a marker of current state. Here we use the 'j' variable instead, but it's almost the same! We reinvented DFA.

The KMP algorithm creates a DFA for each substring to be searched. This is *preprocessing*.

The DFA is then executes on the input string, in the very same fashion like regular expression matcher. (For a DFA used instead of RE[10] matcher, see also another example in this book: 6.2. And also: 15.3.2.)

All the files used

https://smt.st/current_tree/verif/KMP/files_2

5.3.3 The DFA-less version

DFAs generated by the KMP algorithms are *sparse* and have regularities we can observe easily. One popular optimization is not using DFA, but rather a small "partial match" table:

```
// Knuth-Morris-Pratt algorithm
// copypasted from http://cprogramming.com/snippets/source-code/knuthmorrispratt-kmp-
   string-search-algorithm
char *kmp_search(char *haystack, size_t haystack_size, char *needle, size_t
   needle_size)
{
        int *T;
        int i, j;
        char *result = NULL;

        if (needle_size==0)
                return haystack;

        /* Construct the lookup table */
        T = (int*) malloc((needle_size+1) * sizeof(int));
        T[0] = -1;
        for (i=0; i<needle_size; i++)
        {
                T[i+1] = T[i] + 1;
                while (T[i+1] > 0 && needle[i] != needle[T[i+1]-1])
                        T[i+1] = T[T[i+1]-1] + 1;
        }

        printf ("restarts table:\n");
        for (i=0; i<needle_size+1; i++)
                printf ("T[%d]=%d\n", i, T[i]);

        /* Perform the search */
        for (i=j=0; i<haystack_size; )
        {
                if (j>=0)
                        print_compare (haystack, i, needle, j);
                if (j < 0 || haystack[i] == needle[j])
                {
```

[10] Regular Expression

```
                        ++i, ++j;
                        if (j == needle_size)
                        {
                                result = haystack+i-j;
                                break;
                        }
                }
                else
                {
                        j = T[j];
                        if (j==-1)
                                printf ("Restarting needle at the beginning\n");
                        else
                                print_state_needle(needle, j);
                };
        }

        free(T);
        return result;
}
```

Now let's search for 'eel' in the 'eex eel' string:

```
restarts table:
T[0]=-1
T[1]=0
T[2]=1
T[3]=0
going to compare: eex eel
                  eel
                  ^
going to compare: eex eel
                  eel
                   ^
going to compare: eex eel
                  eel
                    ^
Restarting needle at: eel
                       ^
going to compare: eex eel
                   eel
                    ^
Restarting needle at: eel
                      ^
going to compare: eex eel
                    eel
                    ^
Restarting needle at the beginning
going to compare: eex eel
                     eel
                     ^
Restarting needle at the beginning
going to compare: eex eel
                      eel
                      ^
going to compare: eex eel
                      eel
                       ^
going to compare: eex eel
                      eel
                        ^
```

The T[] table is small, which is good for cache memory. But please note: a character is to be loaded twice in case of restart. But this is fine, it's cached anyway at that moment.

Now the interesting thing: if the 'partial match' table is empty (all zeroes), the search function can be implemented naively, like I tried in the first part 5.3.1.

These are the cases of words like 'tesa', 'lee', 'banana'. But a non-empty table must be constructed for words like 'eel', 'oops', 'test', 'elee' – these are words where a prefix (1-character minimum) is repeated in the word. But this is not true for 'banana' – there is no repeating prefix.

All the files used

https://smt.st/current_tree/verif/KMP/files_3

5.3.4 A discussion at HN

https://news.ycombinator.com/item?id=25856558

5.4 Boyer-Moore string search algorithm explanation and formal verification using CBMC

Previously I did this with Knuth-Morris-Pratt algorithm: 5.3.

So again, let's try to reinvent string-search algorithm.

5.4.1 Version 1

If a substring would be compared in reverse order, things may be different. For example, we search for a 'CAT' substring in 'HORSE TURTLE CAT' string.

With naive algorithm, we will first compare 'H' and 'C', we see they are unequal characters and we will advance substring one character ahead:

```
HORSE TURTLE CAT
CAT
  ^
```

Right, but what if we will start at the end of 'CAT' substring proceeding back? We first compare 'R' with 'T' (last character of 'CAT') and we see that they are not equal. We could advance substring one character ahead, but... Can't we see that 'R' character in the middle of 'HORSE' part is absent in the 'CAT' substring? Can't this tell us something useful? Yes – since 'R' is not in 'CAT', we can advance 'CAT' substring more, because we know that this will fail for sure:

```
HORSE TURTLE CAT
 CAT
   ^
```

This too:

```
HORSE TURTLE CAT
  CAT
    ^
```

So if we see a character not in 'CAT', we can advance 3 characters ahead:

```
HORSE TURTLE CAT
   CAT
     ^
```

But this will fail too, also, the space (' ') character is absent in 'CAT' too. OK, we advance 3 characters ahead:

```
HORSE TURTLE CAT
      CAT
        ^
```

No, 'R' is not in 'CAT', advance again 3 characters ahead:

```
HORSE TURTLE CAT
         CAT
           ^
```

Failing again... Adding 3 characters:

```
HORSE TURTLE CAT
            CAT
              ^
```

This is important. 'A' != 'T', but 'A' is in the 'CAT' substring, so this time we advance only 1 character ahead, as in naive search algorithm:

```
HORSE TURTLE CAT
             CAT
               ^
```

Here we will compare 3 characters and return success.

All this is way faster than naive string search algorithm, and is a strong competitor with Knuth-Morris-Pratt algorithm.

This is the source code of a function that is hardcoded to search for 'cat' substring, nothing else:

```
unsigned search_cat_BM_v1 (char *s, unsigned len)
{
        if (len<3)
                return len; // not found
        unsigned read_ops=0;
        unsigned skip; // default skip
        for (unsigned i=0; i<len-2; )
        {
                skip=1;
                read_ops++;
                char char3=s[i+2]; // read operation
                if (char3=='t')
                {
                        read_ops++;
                        char char2=s[i+1]; // read operation
                        if (char2=='a')
                        {
                                read_ops++;
                                char char1=s[i]; // read operation
                                if (char1=='c')
                                {
                                        printf ("%s read_ops=%d\n", __FUNCTION__,
                                           read_ops);
                                        return i; // found
                                }
                                else
                                {
                                        // 1st character isn't 'c'
                                        if ((char1!='c') && (char1!='a') && (char1!='
                                           t'))
                                                skip=3;
                                }
                        }
                        else
                        {
                                // 2nd character isn't 'a'
                                if ((char2!='c') && (char2!='a') && (char2!='t'))
                                        skip=3;
                        }
                }
                else
                {
                        // 3rd character isn't 't'
                        if ((char3!='c') && (char3!='a') && (char3!='t'))
                                skip=3;
                }
                i=i+skip;
        }
        printf ("%s read_ops=%d\n", __FUNCTION__, read_ops);
        return len; // not found
};
```

As compared with a naive brute-force string algorithm, it shows a significant advantage:

```
** cacat
search_cat_brute read_ops=7
search_cat_BM_v1 read_ops=5
** cac cat
search_cat_brute read_ops=10
search_cat_BM_v1 read_ops=5
** cac cac cat
search_cat_brute read_ops=17
search_cat_BM_v1 read_ops=7
** horse turtle cat
search_cat_brute read_ops=16
search_cat_BM_v1 read_ops=8
** horse cat turtle
search_cat_brute read_ops=9
search_cat_BM_v1 read_ops=5
```

5.4.2 Version 2

Now the next observation. If we search for 'CAT' and 'CACAT', we see that 'C' != 'T', but 'C' is in the 'CAT' substring, so we would advance by one character.

```
CACAT
CAT
  ^
```

But... we can also <i>align</i> substring so that the first 'C' character in 'CAT' would match the next 'C' in 'CACAT', by advancing for 2 characters ahead:

```
CACAT
  CAT
    ^
```

We add a rule: "if a mismatched character in string exist in substring, <i>align</i> substring so that they would match" or, in other words, "if 'C' in string isn't equal to the last character in 'CAT' substring, advance by 2 characters":

```
                if (char3=='t')
                {
                        ...
                }
                else
                {
                        // 3rd character isn't 't'
                        if ((char3!='c') && (char3!='a') && (char3!='t'))
                                skip=3;
                        if (char3=='c')         // added
                                skip=2;         // added
                }
                i=i+skip;
```

It may perform even better on some strings:

```
** cacat                           ** cacat
search_cat_brute read_ops=7        search_cat_brute read_ops=7
search_cat_BM_v1 read_ops=5        search_cat_BM_v2 read_ops=4
```

As of aligning 'A' character in the middle of 'CAT' substring – we wouldn't bother, because advancing substring by one character it's the same as is done by default.

Again, our algorithm is hardcoded only for 'CAT'.

And this is the universal algorithm from the R.Sedgewick's book, the first function construct a table of 'skips', the second does the actual search (Java code):

```
    public BoyerMoore(String pat) {
        this.R = 256;
        this.pat = pat;

        // position of rightmost occurrence of c in the pattern
        right = new int[R];
        for (int c = 0; c < R; c++)
            right[c] = -1;
        for (int j = 0; j < pat.length(); j++)
            right[pat.charAt(j)] = j;
    }
...
    public int search(String txt) {
        int m = pat.length();
        int n = txt.length();
        int skip;
        for (int i = 0; i <= n - m; i += skip) {
            skip = 0;
            for (int j = m-1; j >= 0; j--) {
                if (pat.charAt(j) != txt.charAt(i+j)) {
                    skip = Math.max(1, j - right[txt.charAt(i+j)]);
                    break;
                }
            }
            if (skip == 0) return i;    // found
        }
        return n;                       // not found
```

(src)

And what if we search for 'COCOS' substring? What if characters are repeating within search pattern? Well, we will count only rightmost characters, ignoring two first 'CO'.

5.4.3 Formal verification using CBMC

I did that already with my homebrew Knuth-Morris-Pratt algorithm: 5.3.

Listing 5.1: Increase object-bits and unwind options if not enough

```
cbmc --object-bits 9 --trace --function check -DLEN=10 -DCBMC --unwind 9 --unwinding-
    assertions BM_cat_v1.c
cbmc --object-bits 9 --trace --function check -DLEN=10 -DCBMC --unwind 9 --unwinding-
    assertions BM_cat_v2.c
```

```
CBMC version 5.54.0 (cbmc-5.54.0) 64-bit x86_64 linux
Parsing BM_cat_v1.c
Converting
Type-checking BM_cat_v1
Generating GOTO Program
Adding CPROVER library (x86_64)
Removal of function pointers and virtual functions
Generic Property Instrumentation
```

```
Running with 9 object bits, 55 offset bits (user-specified)
Starting Bounded Model Checking
Unwinding loop search_cat_brute.0 iteration 1 file BM_cat_v1.c line 12 function
   search_cat_brute thread 0
Unwinding loop search_cat_brute.0 iteration 2 file BM_cat_v1.c line 12 function
   search_cat_brute thread 0
...
Unwinding loop search_cat_BM_v1.0 iteration 8 file BM_cat_v1.c line 36 function
   search_cat_BM_v1 thread 0
Not unwinding loop search_cat_BM_v1.0 iteration 9 file BM_cat_v1.c line 36 function
   search_cat_BM_v1 thread 0
Runtime Symex: 0.0570656s
size of program expression: 1328 steps
simple slicing removed 2 assignments
Generated 2 VCC(s), 2 remaining after simplification
Runtime Postprocess Equation: 0.000101775s
Passing problem to propositional reduction
converting SSA
Runtime Convert SSA: 0.0178482s
Running propositional reduction
Post-processing
Runtime Post-process: 3.0256e-05s
Solving with MiniSAT 2.2.1 with simplifier
13867 variables, 26286 clauses
SAT checker: instance is UNSATISFIABLE
Runtime Solver: 0.0196347s
Runtime decision procedure: 0.0375839s

** Results:
BM_cat_v1.c function check
[check.assertion.1] line 85 assert: SUCCESS

BM_cat_v1.c function search_cat_BM_v1
[search_cat_BM_v1.unwind.0] line 36 unwinding assertion loop 0: SUCCESS

** 0 of 2 failed (1 iterations)
VERIFICATION SUCCESSFUL
CBMC version 5.54.0 (cbmc-5.54.0) 64-bit x86_64 linux
Parsing BM_cat_v2.c
Converting
Type-checking BM_cat_v2
Generating GOTO Program
Adding CPROVER library (x86_64)
Removal of function pointers and virtual functions
Generic Property Instrumentation
Running with 9 object bits, 55 offset bits (user-specified)
Starting Bounded Model Checking
Unwinding loop search_cat_brute.0 iteration 1 file BM_cat_v2.c line 12 function
   search_cat_brute thread 0
Unwinding loop search_cat_brute.0 iteration 2 file BM_cat_v2.c line 12 function
   search_cat_brute thread 0
...
Unwinding loop search_cat_BM_v2.0 iteration 7 file BM_cat_v2.c line 36 function
   search_cat_BM_v2 thread 0
Unwinding loop search_cat_BM_v2.0 iteration 8 file BM_cat_v2.c line 36 function
   search_cat_BM_v2 thread 0
Not unwinding loop search_cat_BM_v2.0 iteration 9 file BM_cat_v2.c line 36 function
   search_cat_BM_v2 thread 0
Runtime Symex: 0.0558006s
size of program expression: 1364 steps
simple slicing removed 2 assignments
```

```
Generated 2 VCC(s), 2 remaining after simplification
Runtime Postprocess Equation: 0.000102357s
Passing problem to propositional reduction
converting SSA
Runtime Convert SSA: 0.0180041s
Running propositional reduction
Post-processing
Runtime Post-process: 2.8173e-05s
Solving with MiniSAT 2.2.1 with simplifier
14237 variables, 28199 clauses
SAT checker: instance is UNSATISFIABLE
Runtime Solver: 0.0277352s
Runtime decision procedure: 0.0458506s

** Results:
BM_cat_v2.c function check
[check.assertion.1] line 87 assert: SUCCESS

BM_cat_v2.c function search_cat_BM_v2
[search_cat_BM_v2.unwind.0] line 36 unwinding assertion loop 0: SUCCESS

** 0 of 2 failed (1 iterations)
VERIFICATION SUCCESSFUL
```

Now important note.

If you modify these lines so that this conditional expression would compare with other characters...

```
...
{
        // 1st character isn't 'c'
        if ((char1!='c') && (char1!='a') && (char1!='t'))
                skip=3;
}
...
```

CBMC will still verify this algorithm and prove its correctness, because a bug here wouldn't make the algorithm incorrect. It will degrade its performance to a level of naive string search algorithm, as if 'skip=3' here would be replaced with 'skip=1'. It will still work correctly, but slower.

5.4.4 But this algorithm is not finished.

I reimplemented it from R.Sedgewick book.

And he states:

Program BoyerMoore.java implements the bad-character rule part of the Boyer-Moore algorithm. It does not implement the strong good suffix rule.

It coincides with 'delta1' rule from the original Boyer-Moore paper. But there is also a 'delta2' rule, which is just another story for another time.

5.4.5 Files

https://smt.st/current_tree/verif/boyer_moore/files

5.4.6 Discussion at Hacker News

https://news.ycombinator.com/item?id=26900640

Chapter 6

Regular expressions

6.1 KLEE

I've always wanted to generate possible strings for given regular expression. This is not so hard if to dive into regular expression matcher theory and details, but can we force RE matcher to do this?

I took a lightest RE engine I've found: https://github.com/cesanta/slre, and added this:

```
#include "klee.h"

int main(void)
{
#define BUF_SIZE 8
        char s[BUF_SIZE];
        klee_make_symbolic(s, sizeof s, "s");
        s[BUF_SIZE-1]=0;
        if (slre_match("(s|S)t(even|ephen|eve|evie)", s, BUF_SIZE-1, NULL, 0, 0)==
            BUF_SIZE-1)
                 klee_assert(0);
}
```

So I wanted a string that matches the Steven name, in both Steven and Stephen form, also couple diminutive forms (Steve, Stevie). The whole string must have size of 7 characters plus zero byte.

```
% clang -emit-llvm -c -g -O0 -Xclang -disable-O0-optnone -I /tmp/klee_src/include/
    klee/ slre.c

% klee --libc=uclibc --optimize --use-query-log=solver:smt2 slre.bc

...
KLEE: ERROR: slre.c:463: ASSERTION FAIL: 0
KLEE: NOTE: now ignoring this error at this location
...

% ls klee-last | grep err
test000004.assert.err

% ktest-tool klee-last/test000004.ktest | grep data
object 0: data: b'\x00steven\x00'
```

This is indeed the correct string, although, not aligned with start and end of buffer.

Now this RE can match only binary strings that are divisible by 3 (15.3.2):

```
#include "klee.h"

int main(void)
{
#define BUF_SIZE 10
        char s[BUF_SIZE];
        klee_make_symbolic(s, sizeof s, "s");
        s[BUF_SIZE-1]=0;
        if (slre_match("^(1(01*0)*1)*$", s, BUF_SIZE-1, NULL, 0, 0)==BUF_SIZE-1)
                klee_assert(0);
}
```

Got one:

```
% ktest-tool klee-last/test000047.ktest | grep data
object 0: data: b'111110101\xff'
```

It's indeed so, 0b111110101=501, is divisible by 3. Last byte is supposed to be zero-terminating byte, but here is just a noise. OK.

Now, out of whim, let's force to find such a number that two consecutive bits before last one will be both 0:

```
#include "klee.h"

int main(void)
{
#define BUF_SIZE 10
        char s[BUF_SIZE];
        klee_make_symbolic(s, sizeof s, "s");
        s[BUF_SIZE-1]=0;
        if (slre_match("^(1(01*0)*1)*$", s, BUF_SIZE-1, NULL, 0, 0)==BUF_SIZE-1)
                if (s[BUF_SIZE-3]=='0' && s[BUF_SIZE-4]=='0')
                        klee_assert(0);
}
```

```
% ktest-tool klee-last/test000100.ktest | grep data
object 0: data: b'111010001\xff'
```

Indeed, 0b111010001=465, which is divisible by 3.

I tried to force last bit be 0, but KLEE can't find anything – this is understandable. If a last bit is zero, the whole number is even, and can't be divided by 3.

6.2 Fun with regular expressions

Some years ago, when i struggled to understand regular expressions, I found these experiments useful. The following is like my student's notes. I hope, someone will find it useful too.

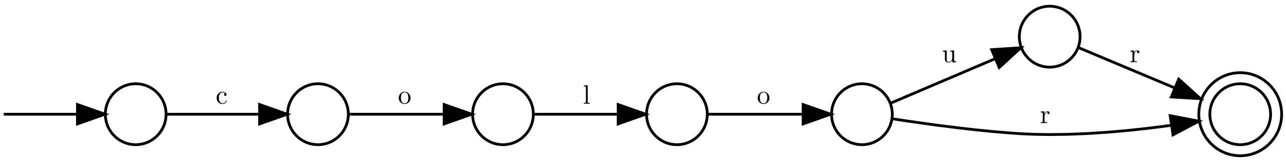
TL;DR: Regular expressions usually used for finding patterns in a string/text and also simple parsing tasks.

6.2.1 colou?r

This is a classic example in many RE tutorials – a RE that matches both U.S. "color" and British "colour".

I'm going to use libfsm, a tool that can produce a DFA for a specific RE.

This is a DFA for it:



But we are programmers, not mathematicians. *libfsm* can also produce a pure C function that will behave as this DFA:

```


int
fsm_main(const char *s)
{
        const char *p;

        enum {
                S0, S1, S2, S3, S4, S5, S6
        } state;

        state = S0;

        for (p = s; *p != '\0'; p++) {
                switch (state) {
                case S0: /* start */
                        switch ((unsigned char) *p) {
                        case 'c': state = S1; break;
                        default:  return -1; /* leaf */
                        }
                        break;

                case S1: /* e.g. "c" */
                        switch ((unsigned char) *p) {
                        case 'o': state = S2; break;
                        default:  return -1; /* leaf */
                        }
                        break;

                case S2: /* e.g. "co" */
                        switch ((unsigned char) *p) {
                        case 'l': state = S3; break;
                        default:  return -1; /* leaf */
                        }
                        break;

                case S3: /* e.g. "col" */
                        switch ((unsigned char) *p) {
                        case 'o': state = S4; break;
                        default:  return -1; /* leaf */
                        }
                        break;

                case S4: /* e.g. "colo" */
                        switch ((unsigned char) *p) {
                        case 'u': state = S5; break;
                        case 'r': state = S6; break;
                        default:  return -1; /* leaf */
                        }
                        break;

                case S5: /* e.g. "colou" */
                        switch ((unsigned char) *p) {
                        case 'r': state = S6; break;
                        default:  return -1; /* leaf */
                        }
                        break;

                case S6: /* e.g. "color" */
                        return -1; /* leaf */

                default:
                        ; /* unreached */
                }
        }

        /* end states */
        switch (state) {
        case S6: return 0x1; /* "colou?r" */
        default: return -1; /* unexpected EOT */
        }
}
```

Now this is important part – this function can be used *in place* of RE matcher. It will work exactly as it. In fact, it emulates DFA for this RE.

It can be used instead of RE matcher in practice, because it works way faster than any RE library. Of course, it has an obvious limitation – it's hardcoded/hardwired to a single RE.

It's very easy to implement DFA in hardware:

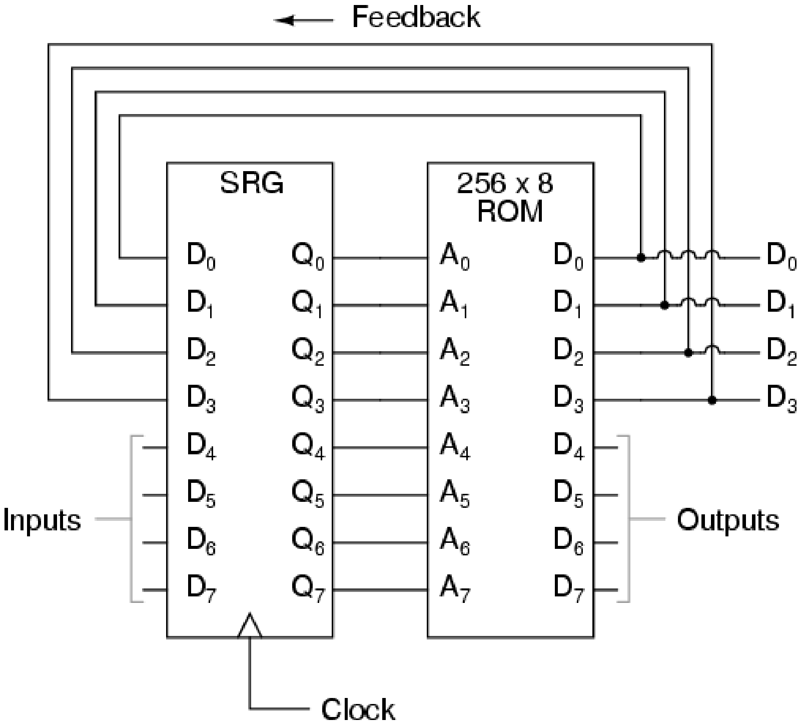


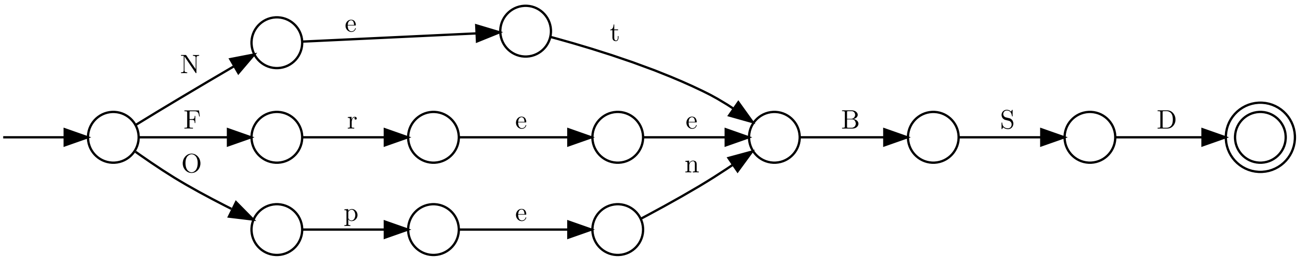
Figure 6.1

(The picture has been shamelessly stolen from this page.)

You see, there is only ROM[1] (as a transition function) and (shift) register. Nothing else. (Please ignore 'outputs' at ROM.) This can be implemented in FPGA[2] or ASIC[3] without any problem. In practice, it can filter a voluminous network traffic for a pattern or RAM[4]: without using a CPU[5] or GPU[6]. So if your FPGA/ASIC is clocked at 1GHz, it can process 10^9 symbols per second, no matter how complex your RE is.

6.2.2 (Net|Open|Free)BSD

... to be matched with main BSD forks.



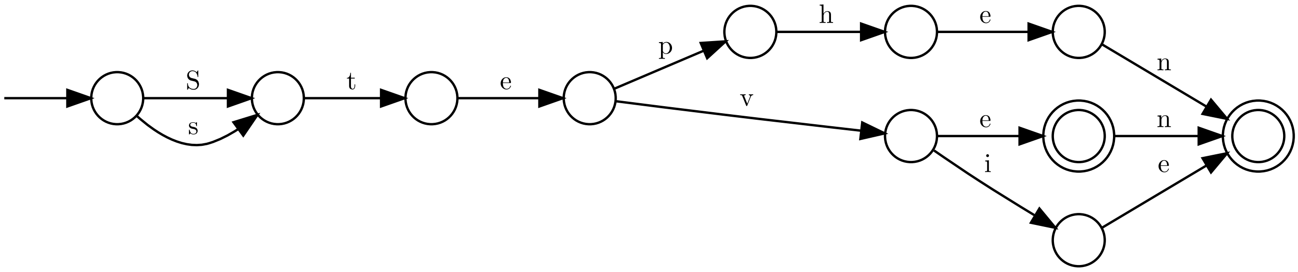
[1] Read-Only Memory
[2] Field-programmable gate array
[3] Application-specific integrated circuit
[4] Random-access memory
[5] Central processing unit
[6] Graphics Processing Unit

```
int
fsm_main(const char *s)
{
        const char *p;

        enum {
                S0, S1, S2, S3, S4, S5, S6, S7, S8, S9,
                S10, S11, S12
        } state;

        state = S0;

        for (p = s; *p != '\0'; p++) {
                switch (state) {
                case S0: /* start */
                        switch ((unsigned char) *p) {
                        case 'N': state = S1; break;
                        case 'O': state = S2; break;
                        case 'F': state = S3; break;
                        default:  return -1; /* leaf */
                        }
                        break;

                case S1: /* e.g. "N" */
                        switch ((unsigned char) *p) {
                        case 'e': state = S12; break;
                        default:  return -1; /* leaf */
                        }
                        break;

                case S2: /* e.g. "O" */
                        switch ((unsigned char) *p) {
                        case 'p': state = S10; break;
                        default:  return -1; /* leaf */
                        }
                        break;

                case S3: /* e.g. "F" */
                        switch ((unsigned char) *p) {
                        case 'r': state = S4; break;
                        default:  return -1; /* leaf */
                        }
                        break;

                case S4: /* e.g. "Fr" */
                        switch ((unsigned char) *p) {
                        case 'e': state = S5; break;
                        default:  return -1; /* leaf */
                        }
                        break;

                case S5: /* e.g. "Fre" */
                        switch ((unsigned char) *p) {
                        case 'e': state = S6; break;
                        default:  return -1; /* leaf */
                        }
                        break;

                case S6: /* e.g. "Net" */
```

```
                        switch ((unsigned char) *p) {
                        case 'B': state = S7; break;
                        default:  return -1; /* leaf */
                        }
                        break;

                case S7: /* e.g. "NetB" */
                        switch ((unsigned char) *p) {
                        case 'S': state = S8; break;
                        default:  return -1; /* leaf */
                        }
                        break;

                case S8: /* e.g. "NetBS" */
                        switch ((unsigned char) *p) {
                        case 'D': state = S9; break;
                        default:  return -1; /* leaf */
                        }
                        break;

                case S9: /* e.g. "NetBSD" */
                        return -1; /* leaf */

                case S10: /* e.g. "Op" */
                        switch ((unsigned char) *p) {
                        case 'e': state = S11; break;
                        default:  return -1; /* leaf */
                        }
                        break;

                case S11: /* e.g. "Ope" */
                        switch ((unsigned char) *p) {
                        case 'n': state = S6; break;
                        default:  return -1; /* leaf */
                        }
                        break;

                case S12: /* e.g. "Ne" */
                        switch ((unsigned char) *p) {
                        case 't': state = S6; break;
                        default:  return -1; /* leaf */
                        }
                        break;

                default:
                        ; /* unreached */
                }
        }

        /* end states */
        switch (state) {
        case S9: return 0x1; /* "(Net|Open|Free)BSD" */
        default: return -1; /* unexpected EOT */
        }
}
```

6.2.3 (s|S)t(even|ephen|eve|evie)

... to be matched with *steven/stephen/steve/stevie*, but also the first letter can be captial.

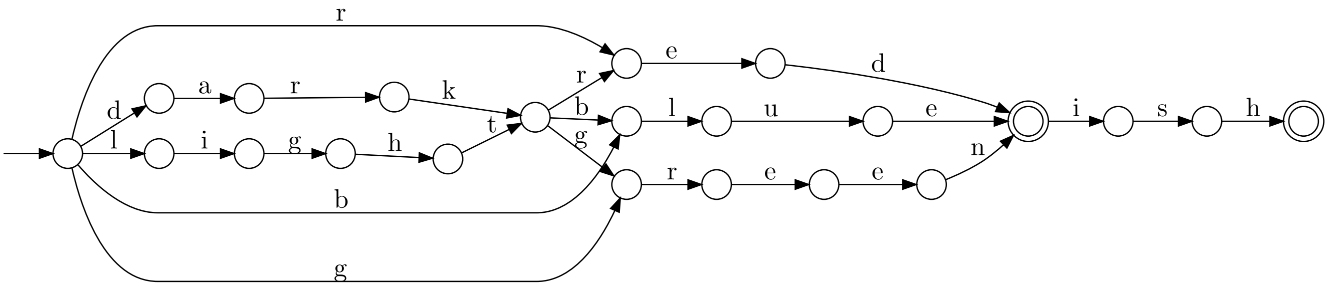


This is important: there are two *accepting states* (in double circles). How it can be handled in C? (States S6 and S8.)

```
int
fsm_main(const char *s)
{
        const char *p;

        enum {
                S0, S1, S2, S3, S4, S5, S6, S7, S8, S9,
                S10
        } state;

        state = S0;

        for (p = s; *p != '\0'; p++) {
                switch (state) {
                case S0: /* start */
                        switch ((unsigned char) *p) {
                        case 'S':
                        case 's': state = S1; break;
                        default:  return -1; /* leaf */
                        }
                        break;

                case S1: /* e.g. "s" */
                        switch ((unsigned char) *p) {
                        case 't': state = S2; break;
                        default:  return -1; /* leaf */
                        }
                        break;

                case S2: /* e.g. "st" */
                        switch ((unsigned char) *p) {
                        case 'e': state = S3; break;
                        default:  return -1; /* leaf */
                        }
                        break;

                case S3: /* e.g. "ste" */
                        switch ((unsigned char) *p) {
                        case 'p': state = S4; break;
                        case 'v': state = S5; break;
                        default:  return -1; /* leaf */
                        }
                        break;

                case S4: /* e.g. "step" */
                        switch ((unsigned char) *p) {
                        case 'h': state = S9; break;
                        default:  return -1; /* leaf */
                        }
                        break;

                case S5: /* e.g. "stev" */
                        switch ((unsigned char) *p) {
                        case 'e': state = S6; break;
                        case 'i': state = S7; break;
                        default:  return -1; /* leaf */
                        }
                        break;

                case S6: /* e.g. "steve" */
                        switch ((unsigned char) *p) {
                        case 'n': state = S8; break;
                        default:  return -1; /* leaf */
                        }
                        break;

                case S7: /* e.g. "stevi" */
                        switch ((unsigned char) *p) {
                        case 'e': state = S8; break;
                        default:  return -1; /* leaf */
                        }
                        break;

                case S8: /* e.g. "stevie" */
                        return -1; /* leaf */

                case S9: /* e.g. "steph" */
                        switch ((unsigned char) *p) {
                        case 'e': state = S10; break;
                        default:  return -1; /* leaf */
                        }
                        break;

                case S10: /* e.g. "stephe" */
                        switch ((unsigned char) *p) {
                        case 'n': state = S8; break;
                        default:  return -1; /* leaf */
                        }
                        break;

                default:
                        ; /* unreached */
                }
        }

        /* end states */
        switch (state) {
        case S6: return 0x1; /* "(s|S)t(even|ephen|eve|evie)" */
        case S8: return 0x1; /* "(s|S)t(even|ephen|eve|evie)" */
        default: return -1; /* unexpected EOT */
        }
}
```

6.2.4 (dark|light)?(red|green|blue)(ish)?

... something more advanced. Here we see a limitation of DFA. (Well, not a limitation.) This DFA is correct. But it will be more (visually) easier to represent it in NFA[7], but this is another story.



BTW[8], *libfsm* can produce a JSON[9] file with all transitions between all the vertices:

```
{
  "statecount": 22,
  "start": 9,
  "end": [ 5, 11 ],
  "edges": [
    { "src": 0, "dst": 2, "symbol": "e" },
    { "src": 1, "dst": 17, "symbol": "u" },
    { "src": 2, "dst": 11, "symbol": "d" },
    { "src": 3, "dst": 4, "symbol": "g" },
    { "src": 4, "dst": 16, "symbol": "h" },
    { "src": 6, "dst": 1, "symbol": "l" },
    { "src": 7, "dst": 6, "symbol": "b" },
    { "src": 7, "dst": 15, "symbol": "g" },
    { "src": 7, "dst": 0, "symbol": "r" },
    { "src": 8, "dst": 3, "symbol": "i" },
    { "src": 9, "dst": 6, "symbol": "b" },
    { "src": 9, "dst": 13, "symbol": "d" },
    { "src": 9, "dst": 15, "symbol": "g" },
    { "src": 9, "dst": 8, "symbol": "l" },
    { "src": 9, "dst": 0, "symbol": "r" },
    { "src": 10, "dst": 21, "symbol": "e" },
    { "src": 11, "dst": 20, "symbol": "i" },
    { "src": 12, "dst": 14, "symbol": "r" },
    { "src": 13, "dst": 12, "symbol": "a" },
    { "src": 14, "dst": 7, "symbol": "k" },
    { "src": 15, "dst": 10, "symbol": "r" },
    { "src": 16, "dst": 7, "symbol": "t" },
    { "src": 17, "dst": 11, "symbol": "e" },
    { "src": 18, "dst": 5, "symbol": "h" },
    { "src": 19, "dst": 11, "symbol": "n" },
    { "src": 20, "dst": 18, "symbol": "s" },
    { "src": 21, "dst": 19, "symbol": "e" }
  ]
}
```

And I wrote a simple utility to enumerate all possible input strings that this DFA will accept, in Racket. It is trivial. It just traverses a DFA recursively down to all accepting states.

```
#lang racket
(require json)

(define cmd (current-command-line-arguments))
```

[7] Nondeterministic Finite Automaton
[8] By The Way
[9] JavaScript Object Notation

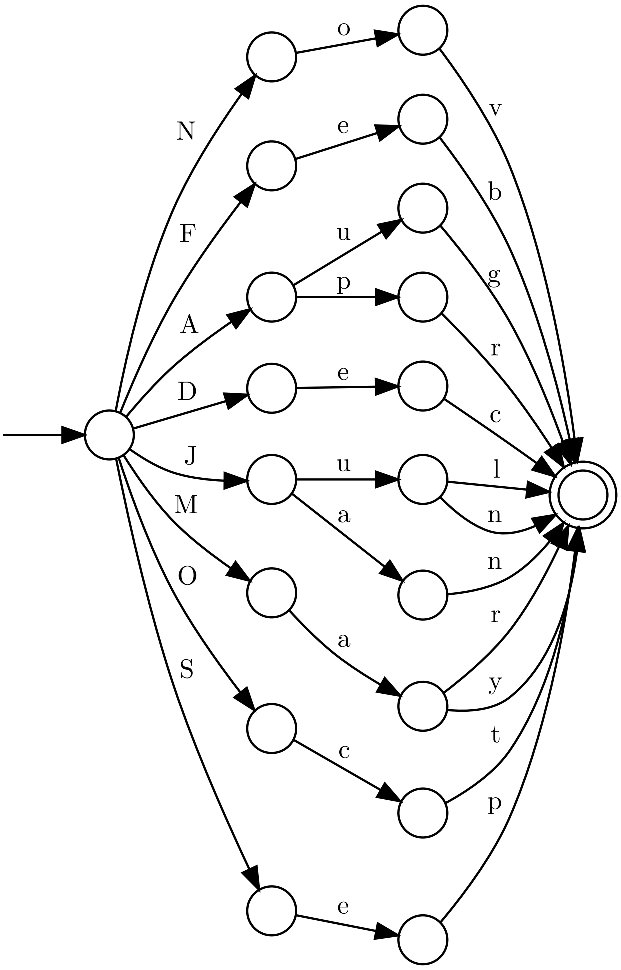
```
(when (= (vector-length cmd) 0)
  (displayln "usage: fname.json")
  (exit))

(define fname (vector-ref cmd 0))

; knobs:
(define traverse-strlen-limit 15)
(define strings-limit 500000)

(define a (read-json (open-input-file fname)))
;a
(define accepted-states (hash-ref a 'end))
(define start-state (hash-ref a 'start))
(define statecount (hash-ref a 'statecount))
(define tbl (make-hash))
(for ([x (hash-ref a 'edges)])
  (let [(src (hash-ref x 'src))
        (dst (hash-ref x 'dst))
        (symbol (hash-ref x 'symbol))]
    (hash-update! tbl src
                  (lambda (x) (cons (list dst symbol) x))
                  '())))

;tbl

(define count 0) ; ah! a global variable

(define (traverse-DAG-recursively tbl start cur-string)
  (when (< (string-length cur-string) traverse-strlen-limit)
    (when (member start accepted-states)
      (displayln (string-append "string matched: [" cur-string "]"))
      (set! count (add1 count))
      (when (> count strings-limit)
        (displayln "strings-limit reached, exiting")
        (exit)))
    (when (hash-has-key? tbl start)
      (for/list ([i (hash-ref tbl start)])
        (traverse-DAG-recursively tbl (first i) (string-append cur-string (second i))
           )))))

(void (traverse-DAG-recursively tbl start-state ""))
(displayln (string-append "total: " (number->string count)))
```

What it will say for that DFA?

```
string matched: [red]
string matched: [redish]
string matched: [lightred]
string matched: [lightredish]
string matched: [lightgreen]
string matched: [lightgreenish]
string matched: [lightblue]
string matched: [lightblueish]
string matched: [green]
string matched: [greenish]
string matched: [darkred]
string matched: [darkredish]
string matched: [darkgreen]
string matched: [darkgreenish]
string matched: [darkblue]
string matched: [darkblueish]
string matched: [blue]
string matched: [blueish]
total: 18
```

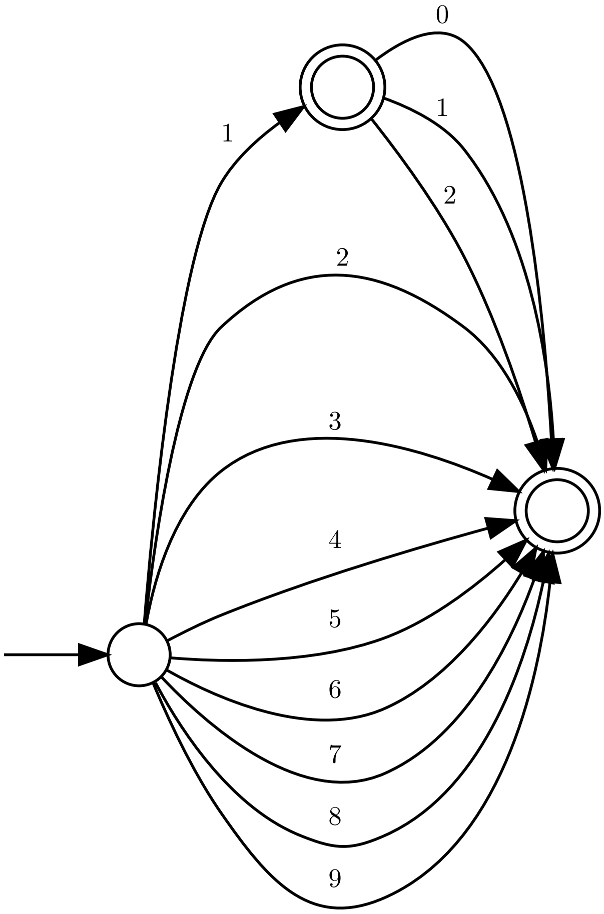
6.2.5 (Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)

In NFA, we would just create 12 paths. Alas, this is DFA, so these paths are overlapped at some vertices:

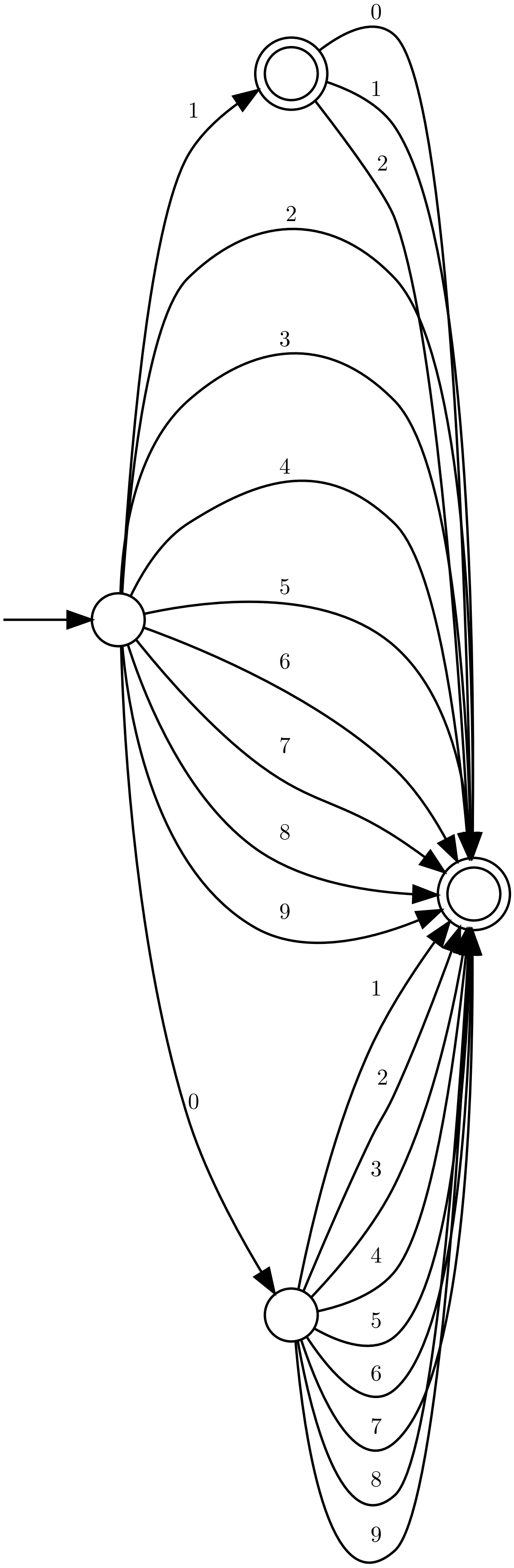


6.2.6 (1|2|3|4|5|6|7|8|9|10|11|12)

'1' is also *accepting state*, but can also lead to '10', '11' and '12':

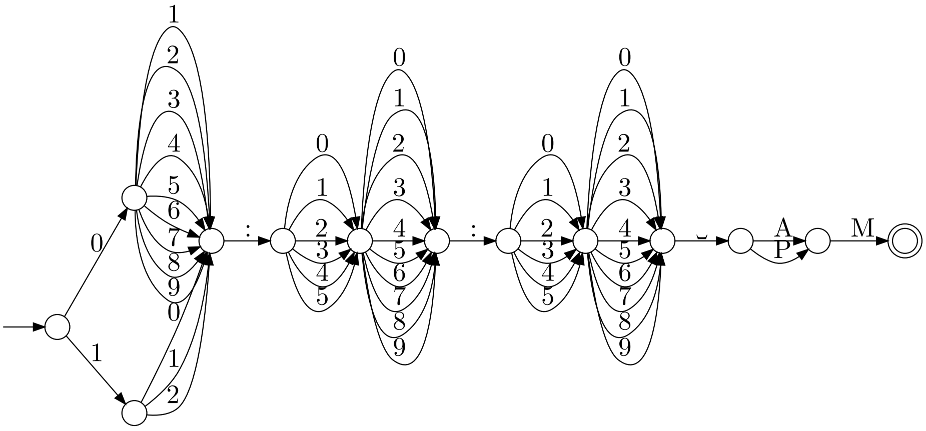

0
1
1
2
2
3
4
5
6
7
8
9

6.2.7 (01|02|03|04|05|06|07|08|09|1|2|3|4|5|6|7|8|9|10|11|12)



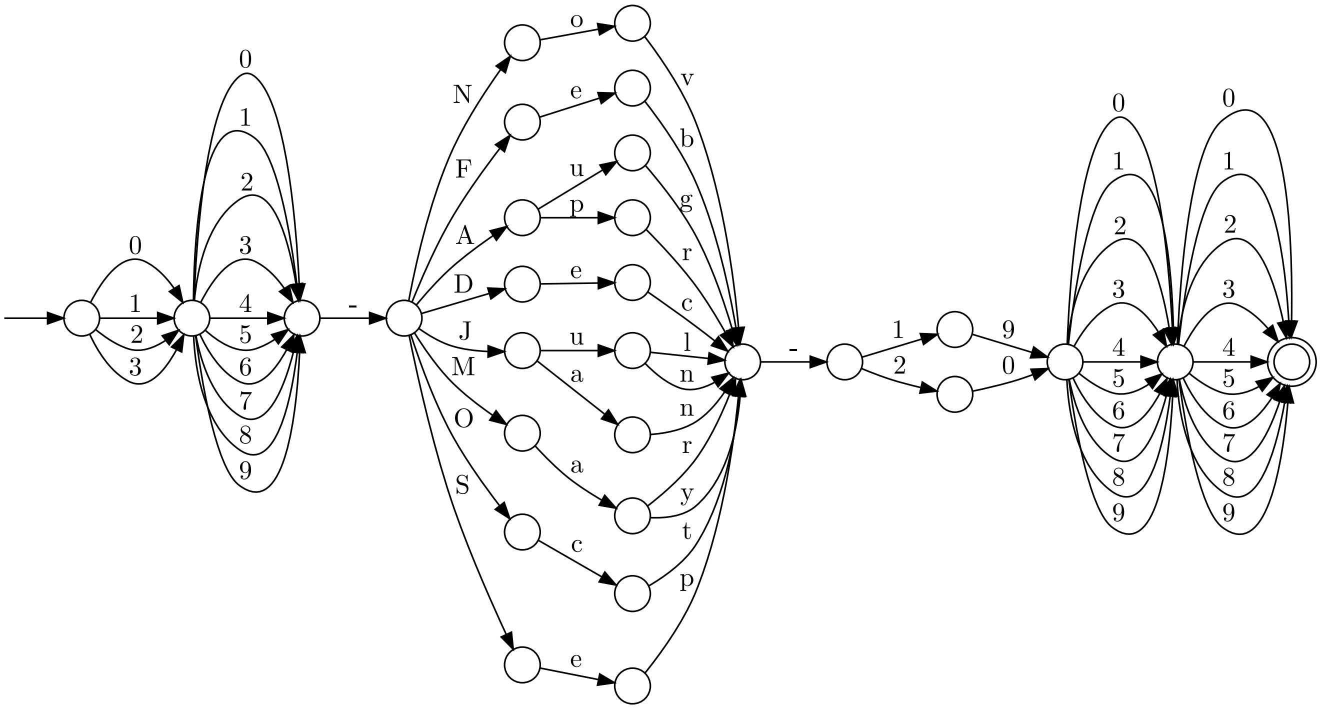
6.2.8 (01|02|03|04|05|06|07|08|09|10|11|12):[0-5][0-9]:[0-5][0-9]␣(A|P)M

(Both hours and minutes are always has two digits.)



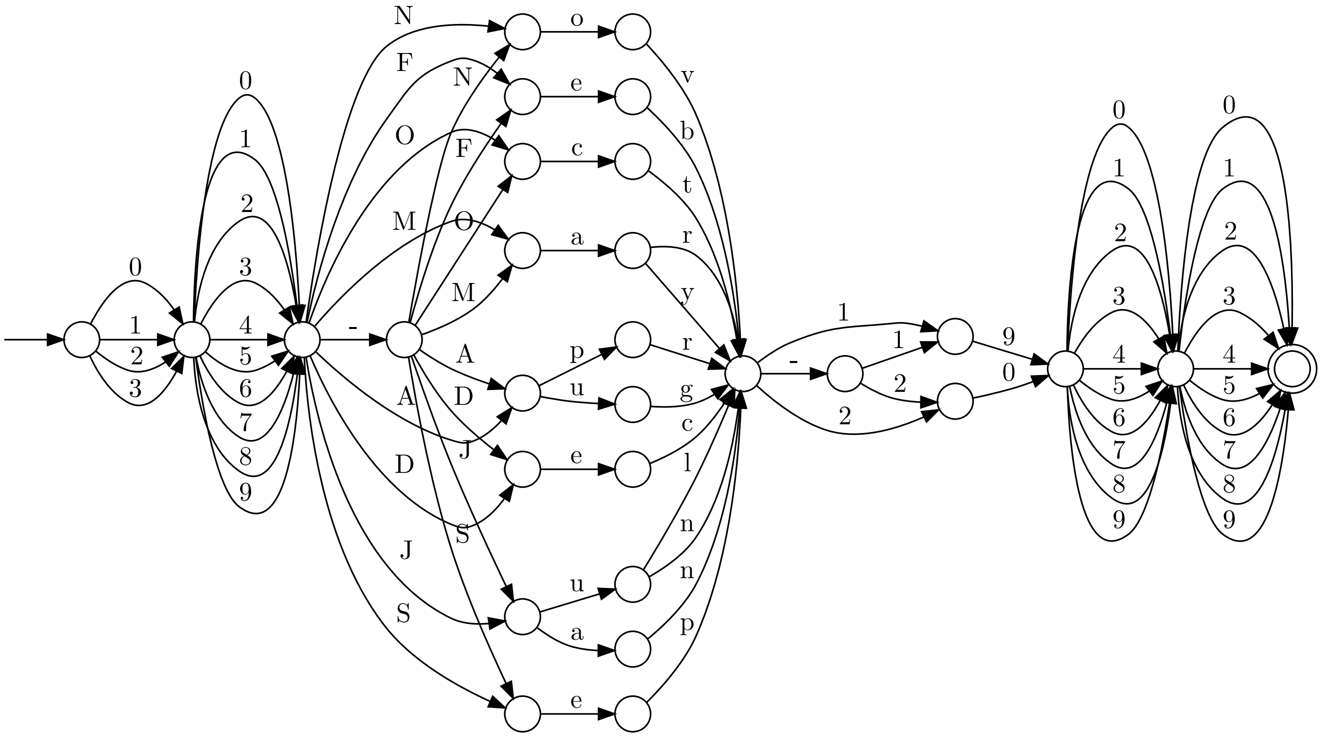
6.2.9 [0123][0-9]-(Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)-(19|20)[0-9]{2}

For the sake of simplicity, only years 19xx and 20xx are accepted:



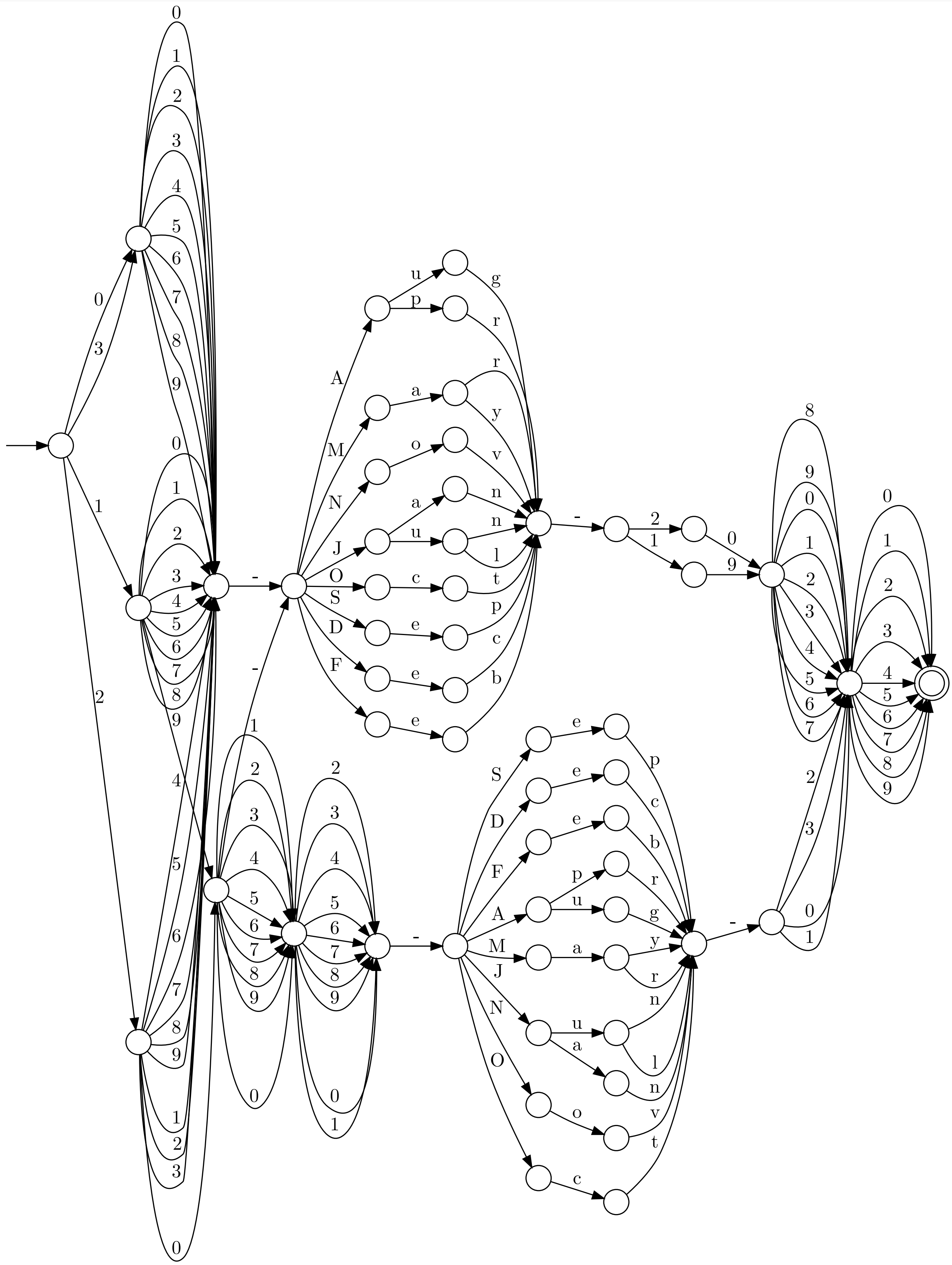
6.2.10 [0123][0-9]-?(Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)-?(19|20)[0-9]{2}

But what if the dash is optional?



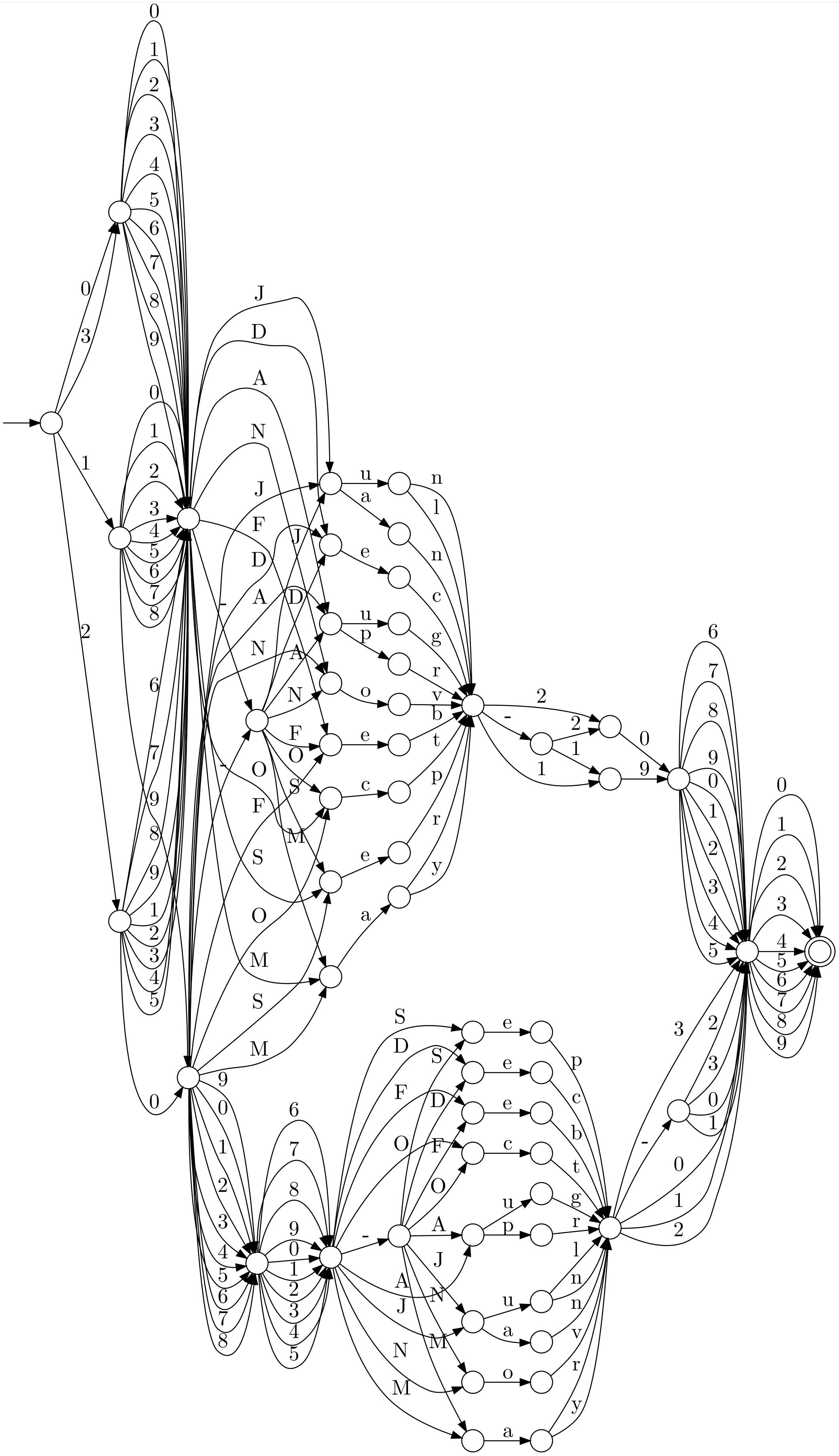
6.2.11 ([0123][0-9]-(Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)-(19|20)[0-9]{2}|(1 9]{2}-(Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)-[0123][0-9])

Can accept dates in DD-MM-YYYY or YYYY-MM-DD format, dash must be present:


0 1 2 3 4 5 6 7 8 9
0
3
1
2
0 1 2 3 4 5 6 7 8 9
4 5 6 7 8 9 1 2 3 0
-
-
1 2 3 4 5 6 7 8 9 0
2 3 4 5 6 7 8 9 0 1
-
A M N J O S D F
u p a o a u c e e e
g r r y v n n l t p c b
-
2 1 0 9
8 9 0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7 8 9
S D F A M J N O
e e e p u a u a o c
p c b r g y r n l n v t
-
2 3 0 1

6.2.12 ([0123][0-9]-?(Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)-?(19|20)[0-9]{2}|(19|20)[0-9]{2}-?(Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)-?[0123][0-9])

Can accept dates in DD-MM-YYYY or YYYY-MM-DD format, a dash is optional. Now the DFA is too large:



Note: using the *standard* RE, it's possible to match only DD-MM-YYYY or DDMMYYYY strings without matching DD-MMYYYY or DDMM-YYYY, but the final DFA would be bulky.

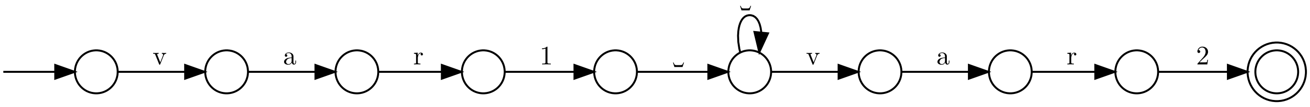
6.2.13 var1␣+var2

Like I stated before, RE often used for parsing. Here are more practical examples.

What it accepts? I'm going to try my utility again:

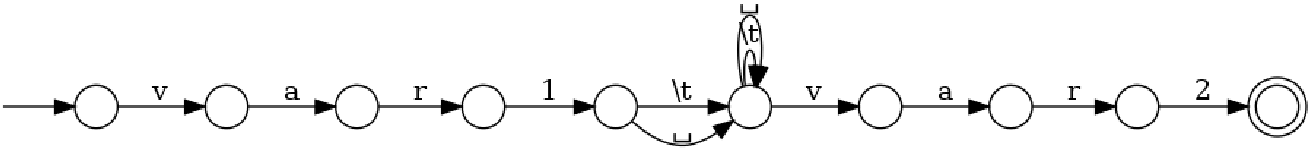
```
string matched: [var1 var2]
string matched: [var1  var2]
string matched: [var1   var2]
string matched: [var1    var2]
string matched: [var1     var2]
string matched: [var1      var2]
total: 6
```

Obviously, a set of input strings is infinite. So my utility has some limit and stopped after it is reached.



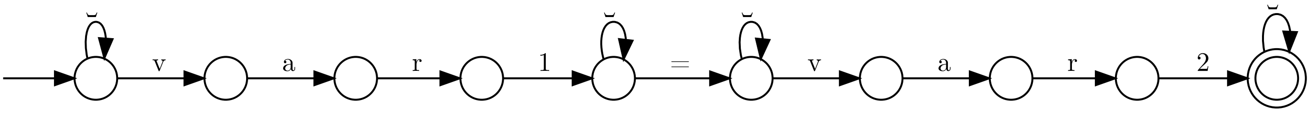
6.2.14 var1[␣\t]+var2

(Extending this RE to whitespaces, i.e., tab symbol.)



6.2.15 ␣*var1␣*=␣*var2␣*

Even more practical. A whitespace can surround input string and the equality sign. Kleene star is used here.

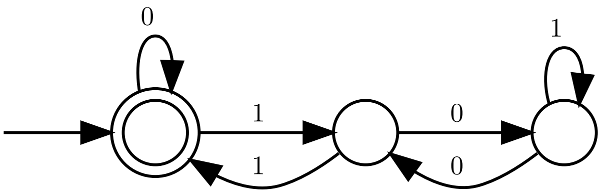


6.2.16 (0|(1(01*0)*1))*

This RE accepts input numbers (in binary form) that are divisible by 3 (15.3.2).

(More about it.)

See also about DFA accepting a binary substring divisible by prime number: 15.3.2.



My utility correctly enumerates such string. But only up to some length...

```
...
string matched: [000110]
string matched: [00011011]
string matched: [0001101111]
string matched: [000110111111]
string matched: [00011011111111]
string matched: [0001101111110]
string matched: [00011011111100]
string matched: [00011011111001]
string matched: [00011011110]
string matched: [0001101111011]
string matched: [00011011110110]
string matched: [000110111100]
string matched: [00011011110011]
...
```

Of course, the number of leading zeroes doesn't affect anything.

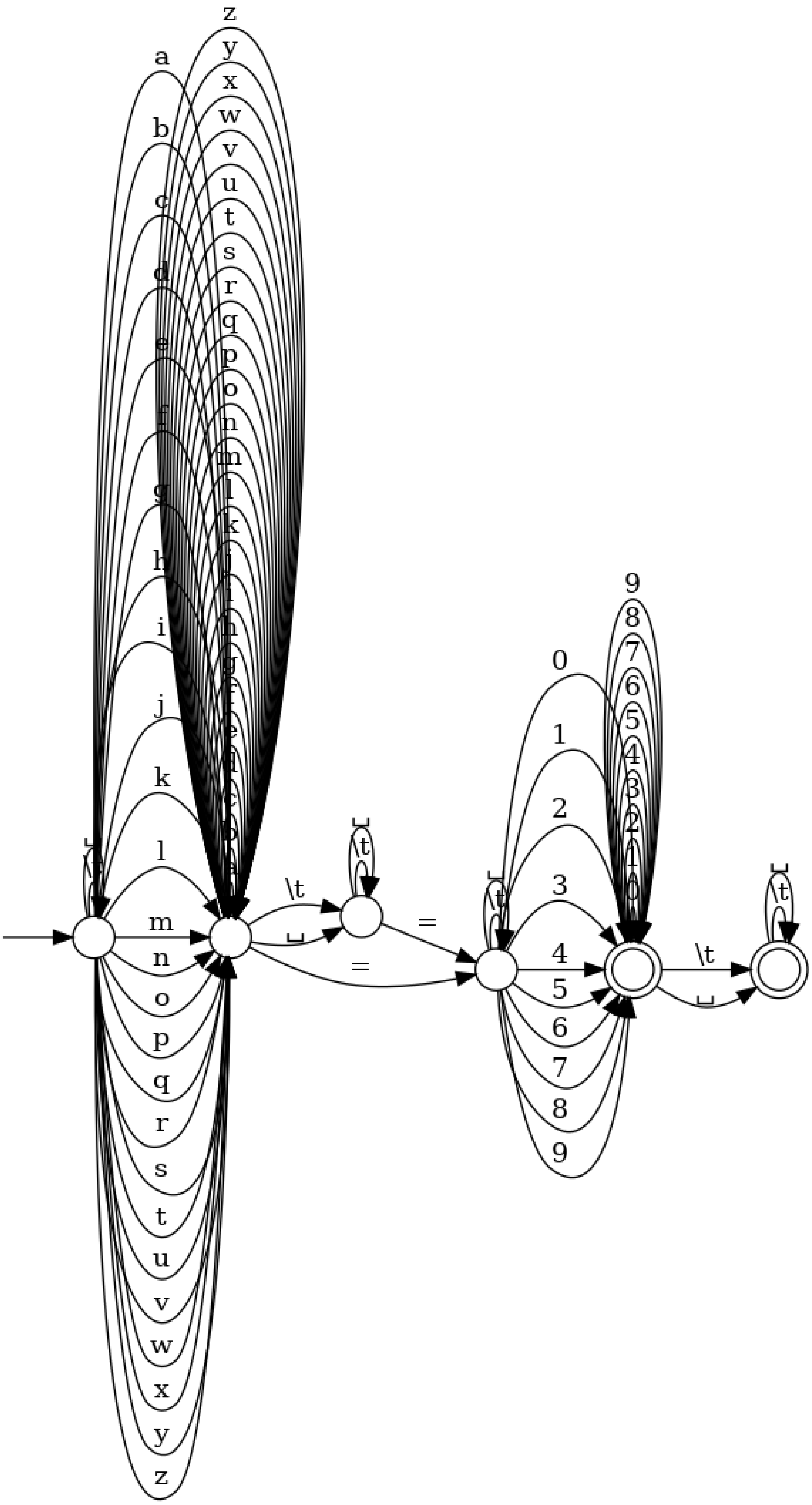
6.2.17 [␣\t]*[a-z]+[␣\t]*=[␣\t]*[0-9]+[␣\t]*

This is the most practically useful example here. It can be used to parse config files in form *option=number*, but whitespaces are also handled – around input string and around the 'equal' sign.

According to my utility, these are (random) sample input strings that can be accepted by the RE:

```
string matched: [zzzzzzzzzzwt=09]
string matched: [zzzzzzzzzztkj=3]
string matched: [zzzzzzzzzzuj=5  ]
string matched: [zzzzzzzzzzy=16 ]
string matched: [zzzzzzzzzzzcr=9]
string matched: [zzzzzzzzzzzxz=2]
string matched: [zzzzzzzzzzuca=2]
string matched: [zzzzzzzzzzyd=44]
string matched: [zzzzzzzzzzuwj=5]
string matched: [zzzzzzzzzzvfj=3]
string matched: [zzzzzzzzzzs     =13]
string matched: [zzzzzzzzzzt     =        8]
string matched: [zzzzzzzzzzuu    =5]
string matched: [zzzzzzzzzzwhi=1]
```

The DFA is complicated by first sight, but it's not:


z
y
x
w
v
u
t
s
r
q
p
o
n
m
l
k
j
i
h
g
f
e
d
c
b
a
a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x
y
z
␣
\t
\t
␣
␣
\t
=
=
␣
\t
0
1
2
3
4
5
6
7
8
9
9
8
7
6
5
4
3
2
1
0
\t
␣
␣
\t

Devising my own parser

Now the fun begins. Can we implement our own parser of such config files without the aid of RE? It turns out, a very good programming exercise.

The first version. Using CBMC I *compare* my version with the pure C DFA simulator produced by *libfsm*.

(I did that already with other algorithms: 5.3, 5.4.)

```
#include <stdio.h>

#include "config.c"

// unwind must be >LEN
// cbmc --trace --function check 1.c --unwind 11 --unwinding-assertions -DCBMC -DLEN
   =4

int naive(char* s)
{
        // skip spaces
        while (*s==' ')
                s++;

        char *tmp=s;
        // read option (a-z)
        for (;;)
        {
                if ((*s>='a') && (*s<='z'))
                {
                        s++;
                        continue;
                }
                else
                        break;
        };

        // skip spaces
        while (*s==' ')
                s++;

        // match '='
        if (*s!='=')
                return -1;
        s++;

        // skip spaces
        while (*s==' ')
                s++;

        tmp=s;
        // read option (0-9)
        for (;;)
        {
                if ((*s>='0') && (*s<='9'))
                {
                        s++;
                        continue;
                }
                else
                        break;
        };

        // skip spaces
        while (*s==' ')
```

```
                s++;

        // all done!
        return 1;
};

#ifdef CBMC
void check()
{
        char s[LEN+1];
        s[LEN]=0;
        __CPROVER_assert (naive(s)==fsm_main(s), "assert");
};
#endif

int main()
{
//#define S " = "
#define S "b="
        printf ("%d\n", naive(S));
        printf ("%d\n", fsm_main(S));
};
```

```
int
fsm_main(const char *s)
{
        const char *p;

        enum {
                S0, S1, S2, S3, S4, S5
        } state;

        state = S0;

        for (p = s; *p != '\0'; p++) {
                switch (state) {
                case S0: /* start */
                        switch ((unsigned char) *p) {
                        case '\t':
                        case ' ': break;
                        case 'a':
                        case 'b':
                        case 'c':
                        case 'd':
                        case 'e':
                        case 'f':
                        case 'g':
                        case 'h':
                        case 'i':
                        case 'j':
                        case 'k':
                        case 'l':
                        case 'm':
                        case 'n':
                        case 'o':
                        case 'p':
                        case 'q':
                        case 'r':
                        case 's':
                        case 't':
```

```
                case 'u':
                case 'v':
                case 'w':
                case 'x':
                case 'y':
                case 'z': state = S1; break;
                default:  return -1; /* leaf */
                }
                break;

        case S1: /* e.g. "a" */
                switch ((unsigned char) *p) {
                case 'a':
                case 'b':
                case 'c':
                case 'd':
                case 'e':
                case 'f':
                case 'g':
                case 'h':
                case 'i':
                case 'j':
                case 'k':
                case 'l':
                case 'm':
                case 'n':
                case 'o':
                case 'p':
                case 'q':
                case 'r':
                case 's':
                case 't':
                case 'u':
                case 'v':
                case 'w':
                case 'x':
                case 'y':
                case 'z': break;
                case '=': state = S2; break;
                case '\t':
                case ' ': state = S3; break;
                default:  return -1; /* leaf */
                }
                break;

        case S2: /* e.g. "a=" */
                switch ((unsigned char) *p) {
                case '\t':
                case ' ': break;
                case '0':
                case '1':
                case '2':
                case '3':
                case '4':
                case '5':
                case '6':
                case '7':
                case '8':
                case '9': state = S4; break;
                default:  return -1; /* leaf */
                }
```

```
                        break;

                case S3: /* e.g. "a\\x09" */
                        switch ((unsigned char) *p) {
                        case '=': state = S2; break;
                        case '\t':
                        case ' ': break;
                        default:  return -1; /* leaf */
                        }
                        break;

                case S4: /* e.g. "a=0" */
                        switch ((unsigned char) *p) {
                        case '0':
                        case '1':
                        case '2':
                        case '3':
                        case '4':
                        case '5':
                        case '6':
                        case '7':
                        case '8':
                        case '9': break;
                        case '\t':
                        case ' ': state = S5; break;
                        default:  return -1; /* leaf */
                        }
                        break;

                case S5: /* e.g. "a=0" */
                        switch ((unsigned char) *p) {
                        case '\t':
                        case ' ': break;
                        default:  return -1; /* leaf */
                        }
                        break;

                default:
                        ; /* unreached */
                }
        }

        /* end states */
        switch (state) {
        case S4: return 0x1; /* "[ \t]*[a-z]+[ \t]*=[ \t]*[0-9]+[ \t]*" */
        case S5: return 0x1; /* "[ \t]*[a-z]+[ \t]*=[ \t]*[0-9]+[ \t]*" */
        default: return -1; /* unexpected EOT */
        }
}
```

What can go wrong? Running CBMC:

```
Trace for check.assertion.1:

State 17 file 1.c line 63 function check thread 0
----------------------------------------------------
  s={ ' ', ' ', 'y', '=', 0 } ({ 00100000, 00100000, 01111001, 00111101, 00000000 })
```

Ouch, my naive parser can't parse the string " y=" correctly. It has no second part (a number) after the 'equality' sign. Probably I can fix it like that:

```
                else
                        break;
```

```
         };
+        // option must be non-empty
+        if (tmp==s)
+                return -1;

         // skip spaces
         while (*s==' ')
@@ -48,6 +51,9 @@
                 else
                         break;
         };
+        // option must be non-empty
+        if (tmp==s)
+                return -1;

         // skip spaces
         while (*s==' ')
```

But it also has problems:

```
Trace for check.assertion.1:

State 17 file 2.c line 69 function check thread 0
----------------------------------------------------
  s={ '\t', 'p', '=', '0', 0 } ({ 00001001, 01110000, 00111101, 00110000, 00000000 })
```

Ouch, I forgot about tabs! By code doesn't handle it.

```
 int naive(char* s)
 {
         // skip spaces
-        while (*s==' ')
+        while ((*s==' ') || (*s=='\t'))
                 s++;

         char *tmp=s;
@@ -27,7 +27,7 @@
                 return -1;

         // skip spaces
-        while (*s==' ')
+        while ((*s==' ') || (*s=='\t'))
                 s++;

         // match '='
@@ -36,7 +36,7 @@
         s++;

         // skip spaces
-        while (*s==' ')
+        while ((*s==' ') || (*s=='\t'))
                 s++;

         tmp=s;
@@ -56,7 +56,7 @@
                 return -1;

         // skip spaces
-        while (*s==' ')
+        while ((*s==' ') || (*s=='\t'))
                 s++;
```

Still problems:

```
Trace for check.assertion.1:

State 17 file 3.c line 69 function check thread 0
----------------------------------------------------
  s={ 'j', '=', '9', -72, 0 } ({ 01101010, 00111101, 00111001, 10111000, 00000000 })
</pre>

<p>
-72 is 0xffffffffffffffb8 or 0xb8, it is non-printable character, anyway.
Fixing it:
</p>

<pre class="normal">
         if (tmp==s)
                 return -1;

-        // skip spaces
-        while ((*s==' ') || (*s=='\t'))
+        // at this point, only spaces and whitespaces are allowed
+        while (*s)
+        {
+                if ((*s!=' ') && (*s!='\t'))
+                {
+                        return -1;
+                };
                 s++;
+        };

         // all done!
         return 1;
```

All cool, up to strings of length 15. It should be noted: 'unwind' parameter must be bigger by at least 1. But not too big, otherwise it will verify your code too slow.

```
 % cbmc --trace --function check 4.c --unwind 16 --unwinding-assertions -DCBMC -DLEN
    =15

...

Solving with MiniSAT 2.2.1 with simplifier
55862 variables, 238864 clauses
SAT checker: instance is UNSATISFIABLE
Runtime decision procedure: 4.23306s

** Results:
[check.assertion.1] assert: SUCCESS
[naive.unwind.0] unwinding assertion loop 0: SUCCESS
[naive.unwind.1] unwinding assertion loop 1: SUCCESS
[naive.unwind.2] unwinding assertion loop 2: SUCCESS
[naive.unwind.3] unwinding assertion loop 3: SUCCESS
[naive.unwind.4] unwinding assertion loop 4: SUCCESS
[naive.unwind.5] unwinding assertion loop 5: SUCCESS
[fsm_main.unwind.0] unwinding assertion loop 0: SUCCESS

** 0 of 8 failed (1 iteration)
VERIFICATION SUCCESSFUL
```

The final source code:

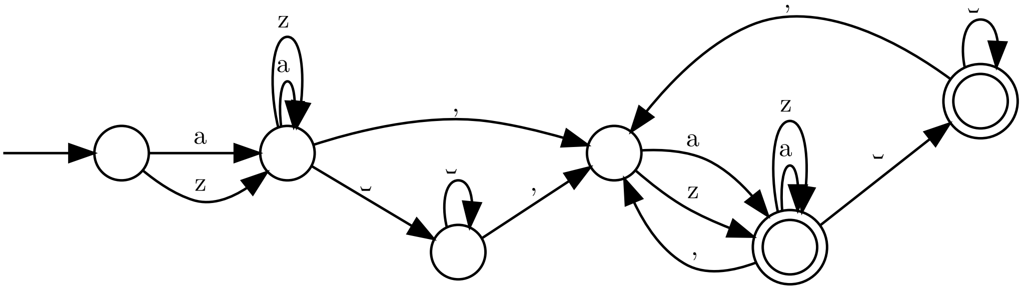
```
#include <stdio.h>

#include "config.c"
```

```
// unwind must be >LEN
// cbmc --trace --function check 4.c --unwind 16 --unwinding-assertions -DCBMC -DLEN
   =15

int naive(char* s)
{
        // skip spaces
        while ((*s==' ') || (*s=='\t'))
                s++;

        char *tmp=s;
        // read option (a-z)
        for (;;)
        {
                if ((*s>='a') && (*s<='z'))
                {
                        s++;
                        continue;
                }
                else
                        break;
        };
        // option must be non-empty
        if (tmp==s)
                return -1;

        // skip spaces
        while ((*s==' ') || (*s=='\t'))
                s++;

        // match '='
        if (*s!='=')
                return -1;
        s++;

        // skip spaces
        while ((*s==' ') || (*s=='\t'))
                s++;

        tmp=s;
        // read option (0-9)
        for (;;)
        {
                if ((*s>='0') && (*s<='9'))
                {
                        s++;
                        continue;
                }
                else
                        break;
        };
        // option must be non-empty
        if (tmp==s)
                return -1;

        // at this point, only spaces and whitespaces are allowed
        while (*s)
        {
                if ((*s!=' ') && (*s!='\t'))
                {
                        return -1;
                };
                s++;
        };

        // all done!
        return 1;
};

#ifdef CBMC
void check()
{
        char s[LEN+1];
        s[LEN]=0;
        __CPROVER_assert (naive(s)==fsm_main(s), "assert");
};
#endif

int main()
{
//#define S " = "
#define S "b="
        printf ("%d\n", naive(S));
        printf ("%d\n", fsm_main(S));
};
```

Obviously, without tools like CBMC, you have a lots of unchecked vulnerable code.

6.2.18 ([az]+␣*,)+[az]+␣*

Even harder RE: for CSV[10] files. For the sake of simplicity, it only accepts strings consisting of 'a' or 'z'.



It's like *nested* or *recursive* RE. But probably too complicated – a more advanced parser should be used for CSV instead of RE.

6.2.19 Syntactic sugar

All the rest is syntactic sugar: "X+" is for "XX*", "X3" is for "XXX", "X1-2" is for "X|XX".

Syntactic sugar is important in practice, but it's better to focus on fundamentals when you start learning.

6.2.20 Homework

All this code is like "grep", it doesn't return *groups* of parsed line, like MM, DD and YYYY separately. You can try hacking what libfsm produces in pure C to add this feature.

Further reading: blog posts about RE by Russ Cox is what I found interesting.

A discussion at HN and reddit.

[10] Comma-Separated Values

Chapter 7

Gray code

7.1 Balanced Gray code and Z3 SMT solver

Suppose, you are making a rotary encoder. This is a device that can signal its angle in some form, like:

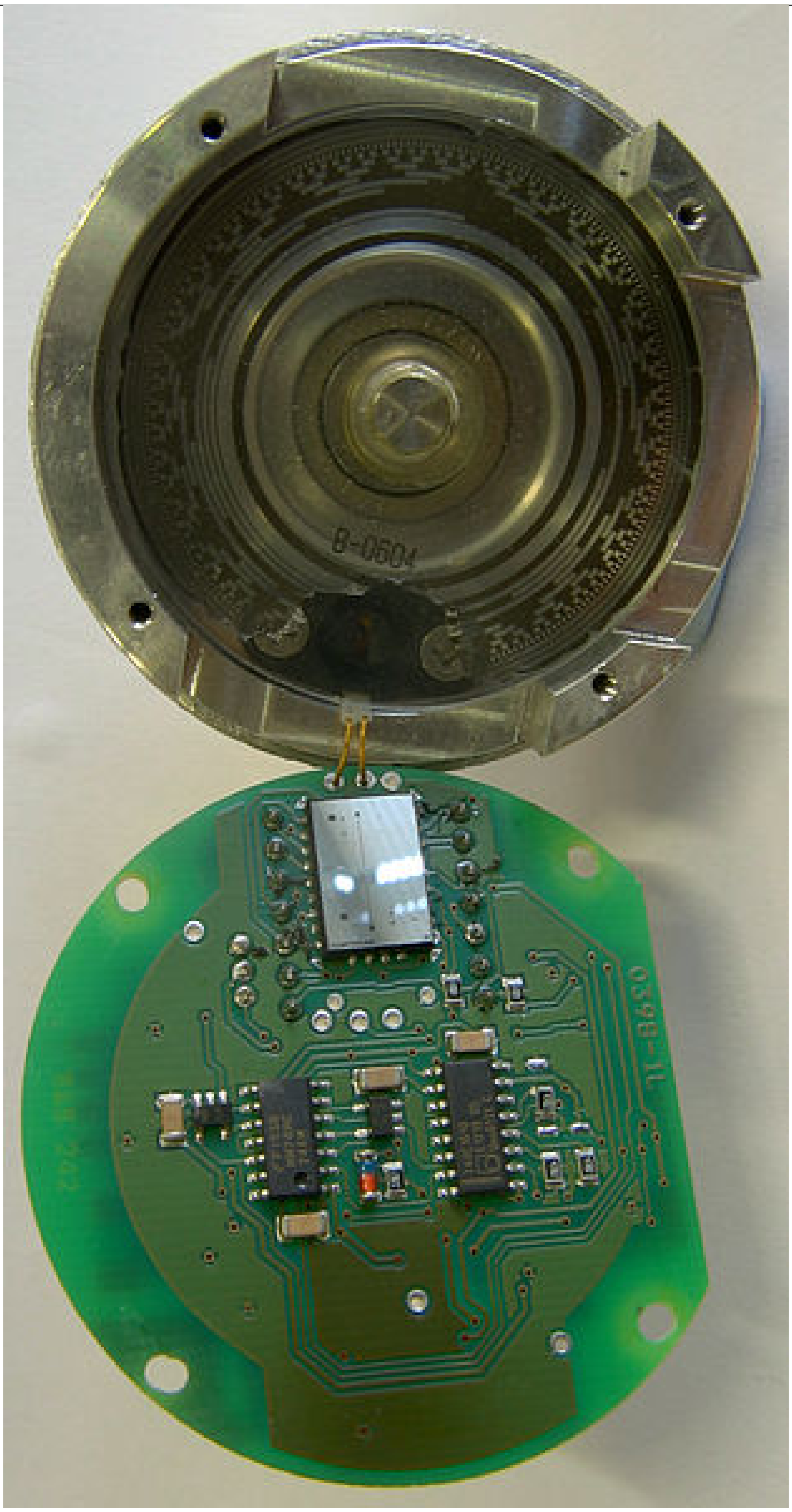


Figure 7.1: Rotary encoder

(The image has been taken from Wikipedia: `https://en.wikipedia.org/wiki/Gray_code`)
Click on bigger image.
This is a rotary (shaft) encoder: `https://en.wikipedia.org/wiki/Rotary_encoder`.

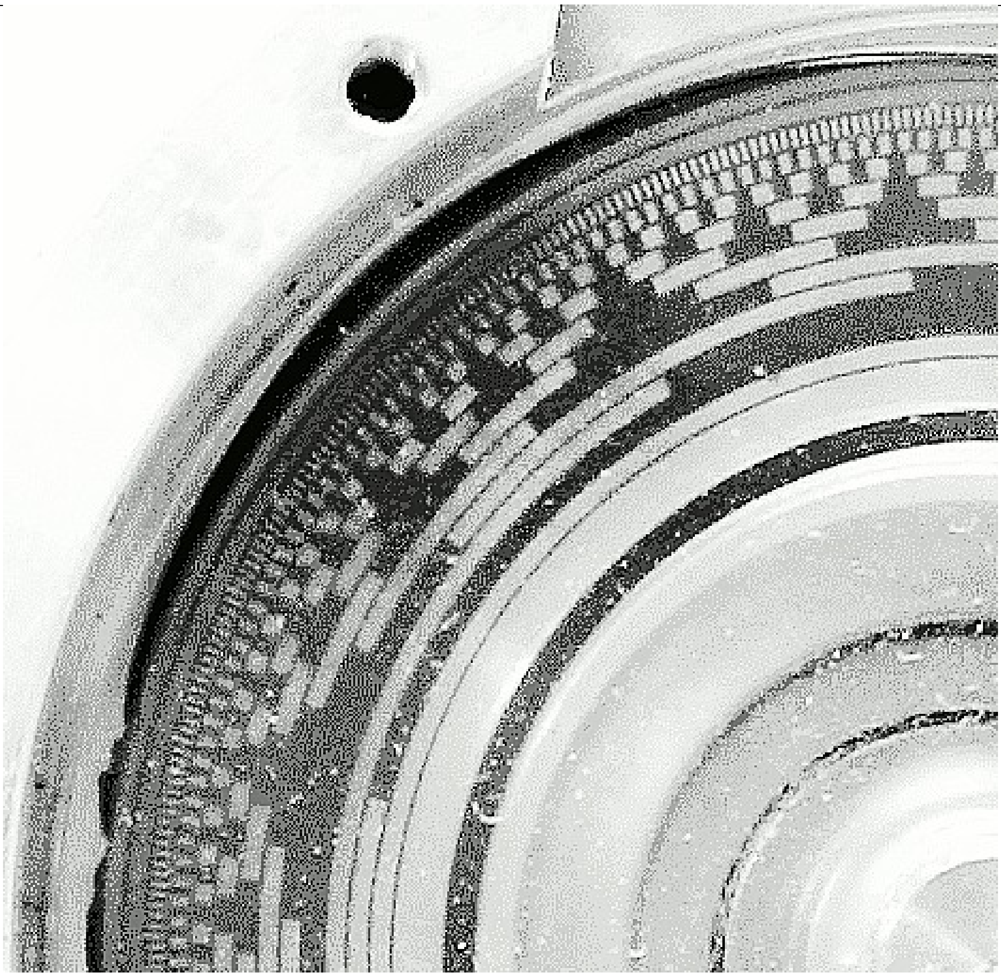


Figure 7.2: Cropped and photoshopped version

(Source: http://homepages.dordt.edu/~ddeboer//S10/304/c_at_d/304S10_RC_TRK.HTM)

Click on bigger one.

There are pins and tracks on rotating wheel. How would you do this? Easiest way is to use binary code. But it has a problem: when a wheel is rotating, in a moment of transition from one state to another, several bits may be changed, hence, undesirable state may be present for a short period of time. This is bad. To deal with it, Gray code was invented: only 1 bit is changed during rotation. Like:

Decimal	Binary	Gray
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

Now the second problem. Look at the picture again. It has a lot of bit changes on the outer circles. And this is electromechanical device. Surely, you may want to make tracks as long as possible, to reduce wearing of both tracks and pins. This is a first problem. The second: wearing should be even across all tracks (this is balanced Gray code).

This is also called: ”They are listed in Gray code or minimum change order, where each subset differs in exactly one element from its neighbors.” (Sriram V. Pemmaraju and Steven Skiena – Computational Discrete Mathematics: Combinatorics and Graph Theory in Mathematica)

How we can find a table for all states using Z3:

```
#!/usr/bin/env python3

from z3 import *

BITS=5

# how many times a run of bits for each bit can be changed (max).
# it can be 4 for 4-bit Gray code or 8 for 5-bit code.
# 12 for 6-bit code (maybe even less)
CHANGES_MAX=8

ROWS=2**BITS
MASK=ROWS-1 # 0x1f for 5 bits, 0xf for 4 bits, etc

def bool_to_int (b):
    if b==True:
        return 1
    return 0

s=Solver()

# add a constraint: Hamming distance between two bitvectors must be 1
# i.e., two bitvectors can differ in only one bit.
# for 4 bits it works like that:
#    s.add(Or(
#        And(a3!=b3,a2==b2,a1==b1,a0==b0),
#        And(a3==b3,a2!=b2,a1==b1,a0==b0),
#        And(a3==b3,a2==b2,a1!=b1,a0==b0),
#        And(a3==b3,a2==b2,a1==b1,a0!=b0)))
def hamming1(l1, l2):
    assert len(l1)==len(l2)
    r=[]
    for i in range(len(l1)):
        t=[]
        for j in range(len(l1)):
            if i==j:
                t.append(l1[j]!=l2[j])
            else:
                t.append(l1[j]==l2[j])
        r.append(And(t))
    s.add(Or(r))

# add a constraint: bitvectors must be different.
# for 4 bits works like this:
#    s.add(Or(a3!=b3, a2!=b2, a1!=b1, a0!=b0))
def not_eq(l1, l2):
    assert len(l1)==len(l2)
    t=[l1[i]!=l2[i] for i in range(len(l1))]
    s.add(Or(t))

code=[[Bool('code_%d_%d' % (r,c)) for c in range(BITS)] for r in range(ROWS)]
ch=[[Bool('ch_%d_%d' % (r,c)) for c in range(BITS)] for r in range(ROWS)]

# each rows must be different from a previous one and a next one by 1 bit:
for i in range(ROWS):
    # get bits of the current row:
```

```
    lst1=[code[i][bit] for bit in range(BITS)]
    # get bits of the next row.
    # important: if the current row is the last one, (last+1)&MASK==0, so we overlap here:
    lst2=[code[(i+1)&MASK][bit] for bit in range(BITS)]
    hamming1(lst1, lst2)

# no row must be equal to any another row:
for i in range(ROWS):
    for j in range(ROWS):
        if i==j:
            continue
        lst1=[code[i][bit] for bit in range(BITS)]
        lst2=[code[j][bit] for bit in range(BITS)]
        not_eq(lst1, lst2)

# 1 in ch[] table means that run of 1's has been changed to run of 0's, or back.
# "run" change detected using simple XOR:
for i in range(ROWS):
    for bit in range(BITS):
        # row overlapping works here as well:
        s.add(ch[i][bit]==Xor(code[i][bit],code[(i+1)&MASK][bit]))

# only CHANGES_MAX of 1 bits is allowed in ch[] table for each bit:
for bit in range(BITS):
    t=[ch[i][bit] for i in range(ROWS)]
    # this is a dirty hack.
    # AtMost() takes arguments like:
    # AtMost(v1, v2, v3, v4, 2) <- this means, only 2 booleans (or less) from the list can be True.
    # but we need to pass a list here.
    # so a CHANGES_MAX number is appended to a list and a new list is then passed as arguments list:
    s.add(AtMost(*(t+[CHANGES_MAX])))

result=s.check()
if result==unsat:
    exit(0)
m=s.model()

# get the model.

print ("code table:")

for i in range(ROWS):
    t=""
    for bit in range(BITS):
        # comma at the end means "no newline":
        t=t+str(bool_to_int(is_true(m[code[i][BITS-1-bit]])))+" "
    print (t)

print ("ch table:")

stat={}

for i in range(ROWS):
    t=""
    for bit in range(BITS):
        x=is_true(m[ch[i][BITS-1-bit]])
        if x:
            # increment if bit is present in dict, set 1 if not present
            stat[bit]=stat.get(bit, 0)+1
        # comma at the end means "no newline":
        t=t+str(bool_to_int(x))+" "
    print (t)

print ("stat (bit number: number of changes): ", stat)
```

(The source code: https://smt.st/current_tree/gray_code/SMT/gray.py)

For 4 bits, 4 changes is enough:

```
code table:
0 1 0 1
0 0 0 1
0 0 1 1
0 0 1 0
1 0 1 0
1 0 1 1
1 1 1 1
1 1 0 1
1 0 0 1
1 0 0 0
0 0 0 0
0 1 0 0
1 1 0 0
1 1 1 0
0 1 1 0
0 1 1 1
ch table:
0 1 0 0
0 0 1 0
0 0 0 1
1 0 0 0
0 0 0 1
0 1 0 0
0 0 1 0
0 1 0 0
0 0 0 1
1 0 0 0
0 1 0 0
1 0 0 0
0 0 1 0
1 0 0 0
0 0 0 1
0 0 1 0
stat (bit number: count of changes):  {0: 4, 1: 4, 2: 4, 3: 4}
```

8 changes for 5 bits: https://smt.st/current_tree/gray_code/SMT/5.txt. 12 for 6 bits (or maybe even less): https://smt.st/current_tree/gray_code/SMT/6.txt.

7.1.1 Duke Nukem 3D from 1990s

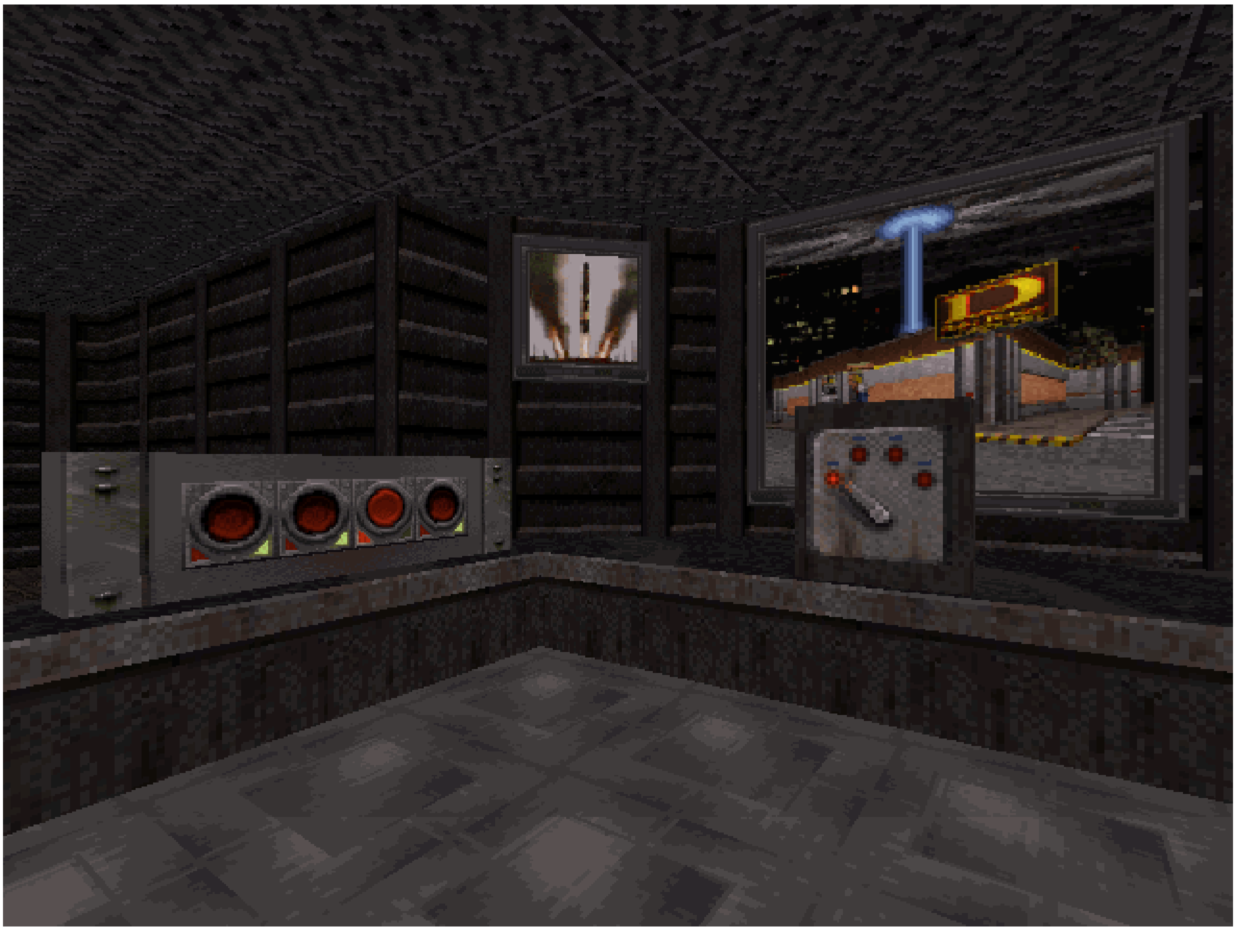


Figure 7.3: Duke Nukem 3D

Another application of Gray code:

```
with.inspiring@flair.and.erudition (Mike Naylor) wrote:

>In Duke Nukem, you often come upon panels that have four buttons in a row,
>all in their "off" position. Each time you "push" a button, it toggles from
>one state to the other. The object is to find the unique combination that
>unlocks something in the game.

>My question is: What is the most efficient order in which to push the
>buttons so that every combination is tested with no wasted effort?

A Gray Code. :-)

(Oh, you wanted to know what one would be?  How about:
0000
0001
0011
0010
0110
0111
0101
0100
1100
1101
1111
1110
```

```
1010
1000
1001
1011

Or, if you prefer, with buttons A,B,C,D: D,C,D,B,D,C,D,A,D,C,D,B,C,D,C
It isn't the "canonical" Gray code (or if it is, it is by Divine
Providence), but it works.

Douglas Limmer -- lim...@math.orst.edu
"No wonder these mathematical wizards were nuts - went off the beam -
he'd be pure squirrel-food if he had half that stuff in _his_ skull!"
E. E. "Doc" Smith, _Second Stage Lensmen_
```

(https://groups.google.com/forum/#!topic/rec.puzzles/Dh2H-pGJcbI)

Obviously, using our solution, you can minimize all movements in this ancient videogame, for 4 switches, that would be 4*4=16 switches. With our solution (balanced Gray code), wearing would be even across all 4 switches.

7.1.2 Towers of Hanoi

"The standard n-bit Gray code gives a solution to the Towers of Hanoi problem with n levels. The position of the bit that changes tells us which level of the tower we must move." (http://www.maths.liv.ac.uk/~mathsclub/talks/20160130/talk1/joel_summary.pdf).

7.2 Gray code in MaxSAT

This is remake of gray code generator for Z3 (7.1).

Here is also *ch[]* table, but we add soft clauses for it here. The goal is to make as many *False*'s in *ch[]* table, as possible.

```
#!/usr/bin/env python3

import subprocess, os, itertools, my_utils, SAT_lib

BITS=6

# how many times a run of bits for each bit can be changed (max).
# it can be 4 for 4-bit Gray code or 8 for 5-bit code.
# 12 for 6-bit code (maybe even less)

ROWS=2**BITS
MASK=ROWS-1 # 0x1f for 5 bits, 0xf for 4 bits, etc

def do_all():
    s=SAT_lib.SAT_lib(maxsat=True)

    code=[s.alloc_BV(BITS) for r in range(ROWS)]
    ch=[s.alloc_BV(BITS) for r in range(ROWS)]

    # each rows must be different from a previous one and a next one by 1 bit:
    for i in range(ROWS):
        # get bits of the current row:
        lst1=[code[i][bit] for bit in range(BITS)]
        # get bits of the next row.
        # important: if the current row is the last one, (last+1)&MASK==0, so we
            overlap here:
        lst2=[code[(i+1)&MASK][bit] for bit in range(BITS)]
        s.hamming1(lst1, lst2)

    # no row must be equal to any another row:
    for i in range(ROWS):
        for j in range(ROWS):
            if i==j:
                continue
            lst1=[code[i][bit] for bit in range(BITS)]
            lst2=[code[j][bit] for bit in range(BITS)]
            s.fix_BV_NEQ(lst1, lst2)

    # 1 in ch[] table means that run of 1's has been changed to run of 0's, or back.
    # "run" change detected using simple XOR:
    for i in range(ROWS):
        for bit in range(BITS):
            # row overlapping works here as well.
            # we add here "soft" constraint with weight=1:
            s.fix_soft(s.EQ(ch[i][bit], s.XOR(code[i][bit],code[(i+1)&MASK][bit])),
               False, weight=1)

    if s.solve()==False:
        print ("unsat")
        exit(0)

    print ("code table:")

    for i in range(ROWS):
        tmp=""
        for bit in range(BITS):
            t=s.get_var_from_solution(code[i][BITS-1-bit])
            if t:
                tmp=tmp+"*"
            else:
                tmp=tmp+" "
        print (tmp)

    # get statistics:
    stat={}

    for i in range(ROWS):
        for bit in range(BITS):
            x=s.get_var_from_solution(ch[i][BITS-1-bit])
            if x==0:
                # increment if bit is present in dict, set 1 if not present
                stat[bit]=stat.get(bit, 0)+1

    print ("stat (bit number: number of changes): ")
    print (stat)

do_all()
```

So it does, for 5-bit Gray code:

```
code table:
 ****
*****
* ***
  ***
  **
 ***
 **
 *
**
**  *
*** *
***
```

```
****
* **
*  *
*
*   *
* * *
* *
  *
  * *
    *

   *
   **
*  **
** **
** *
 * *
 * **
 *  *
 ** *
stat (bit number: number of changes):
{0: 6, 1: 4, 2: 6, 3: 6, 4: 10}
```

... for 6-bit Gray code:

```
code table:
*   **
*   *
*  **
*  *
   *
  **
  ** *
* ** *
* **
* *
***
****
** *
 * *
 ***
 ****
  ***
* ***
* * *
*** *
*****
******
 *****
 ** **
*** **
* * **
* *  *
***  *
**** *
 *** *
 **  *
  *  *
  * **
  * *
 ** *
 **
```

```
  *

     *
   * *
 * * *
 *   *
 *
 *  *
    *
   **
 * **
 * ***
   ***
  ****
* ****
*  ***
** ***
** **
**  *
**
*
*    *
*  * *
** * *
**   *
**  **
 *  **
    **
stat (bit number: number of changes):
{5: 10, 3: 14, 4: 8, 0: 10, 2: 6, 1: 16}
```

7.3 Further reading

- Martin Gardner – Knotted Doughnuts and Other Mathematical Entertainments, page 22.
- Donald Knuth – TAOCP, PRE-FASCICLE 2A[1].
- Jörg Arndt – Matters Computational[2], 1.16, *Gray code and parity*.

[1] https://yurichev.com/mirrors/Donald%20Knuth/TAOCP%202a%207.2.1.1/fasc2a.pdf

[2] https://www.jjj.de/fxt/fxtbook.pdf

Chapter 8

Recreational mathematics and puzzles

8.1 Sudoku

Sudoku puzzle is a 9*9 grid with some cells filled with values, some are empty:

		5	3					
8							2	
	7			1		5		
4					5	3		
	1			7				6
		3	2				8	
	6		5					9
		4					3	
					9	7		

Unsolved Sudoku

Numbers of each row must be unique, i.e., it must contain all 9 numbers in range of 1..9 without repetition. Same story for each column and also for each 3*3 square.

This puzzle is good candidate to try SMT solver on, because it's essentially an unsolved system of equations.

8.1.1 Simple sudoku in SMT

The first idea

The only thing we must decide is that how to determine in one expression, if the input 9 variables has all 9 unique numbers? They are not ordered or sorted, after all.

From the school-level arithmetic, we can devise this idea:

$$\underbrace{10^{i_1} + 10^{i_2} + \cdots + 10^{i_9}}_{9} = 1111111110 \tag{8.1}$$

Take each input variable, calculate 10^i and sum them all. If all input values are unique, each will be settled at its own place. Even more than that: there will be no holes, i.e., no skipped values. So, in case of Sudoku, 1111111110 number will be final result, indicating that all 9 input values are unique, in range of 1..9.

Exponentiation is heavy operation, can we use binary operations? Yes, just replace 10 with 2:

$$\underbrace{2^{i_1} + 2^{i_2} + \cdots + 2^{i_9}}_{9} = 1111111110_2 \tag{8.2}$$

The effect is just the same, but the final value is in base 2 instead of 10.

Now a working example:

```
#!/usr/bin/env python3

import sys
from z3 import *

"""
coordinates:
------------------------------
00 01 02 | 03 04 05 | 06 07 08
10 11 12 | 13 14 15 | 16 17 18
20 21 22 | 23 24 25 | 26 27 28
------------------------------
30 31 32 | 33 34 35 | 36 37 38
40 41 42 | 43 44 45 | 46 47 48
50 51 52 | 53 54 55 | 56 57 58
------------------------------
60 61 62 | 63 64 65 | 66 67 68
70 71 72 | 73 74 75 | 76 77 78
80 81 82 | 83 84 85 | 86 87 88
------------------------------
"""

s=Solver()

# using Python list comprehension, construct array of arrays of BitVec instances:
cells=[[BitVec('cell%d%d' % (r, c), 16) for c in range(9)] for r in range(9)]

# http://www.norvig.com/sudoku.html
# http://www.mirror.co.uk/news/weird-news/worlds-hardest-sudoku-can-you-242294
puzzle="
    ..53.....8......2..7..1.5..4....53...1..7...6..32...8..6.5....9..4....3......97.."

# process text line:
current_column=0
current_row=0
for i in puzzle:
        if i!='.':
                s.add(cells[current_row][current_column]==BitVecVal(int(i),16))
        current_column=current_column+1
        if current_column==9:
                current_column=0
                current_row=current_row+1

one=BitVecVal(1,16)
mask=BitVecVal(0b1111111110,16)

# for all 9 rows
for r in range(9):
        s.add(((one<<cells[r][0]) +
                (one<<cells[r][1]) +
                (one<<cells[r][2]) +
                (one<<cells[r][3]) +
                (one<<cells[r][4]) +
                (one<<cells[r][5]) +
                (one<<cells[r][6]) +
                (one<<cells[r][7]) +
                (one<<cells[r][8]))==mask)

# for all 9 columns
for c in range(9):
        s.add(((one<<cells[0][c]) +
                (one<<cells[1][c]) +
                (one<<cells[2][c]) +
                (one<<cells[3][c]) +
                (one<<cells[4][c]) +
                (one<<cells[5][c]) +
                (one<<cells[6][c]) +
                (one<<cells[7][c]) +
                (one<<cells[8][c]))==mask)

# enumerate all 9 squares
for r in range(0, 9, 3):
        for c in range(0, 9, 3):
                # add constraints for each 3*3 square:
                        s.add((one<<cells[r+0][c+0]) +
                                (one<<cells[r+0][c+1]) +
                                (one<<cells[r+0][c+2]) +
                                (one<<cells[r+1][c+0]) +
                                (one<<cells[r+1][c+1]) +
                                (one<<cells[r+1][c+2]) +
                                (one<<cells[r+2][c+0]) +
                                (one<<cells[r+2][c+1]) +
                                (one<<cells[r+2][c+2])==mask)

print (s.check())
#print (s.model())
m=s.model()

for r in range(9):
        for c in range(9):
                sys.stdout.write (str(m[cells[r][c]])+" ")
        print ("")
```

(https://smt.st/current_tree/puzzles/sudoku/1/sudoku_plus_Z3.py)

```
% time python sudoku_plus_Z3.py
1 4 5 3 2 7 6 9 8
8 3 9 6 5 4 1 2 7
6 7 2 9 1 8 5 4 3
4 9 6 1 8 5 3 7 2
2 1 8 4 7 3 9 5 6
7 5 3 2 9 6 4 8 1
3 6 7 5 4 2 8 1 9
9 8 4 7 6 1 2 3 5
5 2 1 8 3 9 7 6 4

real    0m11.717s
user    0m10.896s
sys     0m0.068s
```

Even more, we can replace summing operation to logical OR:

$$\underbrace{2^{i_1} \vee 2^{i_2} \vee \cdots \vee 2^{i_9}}_{9} = 1111111110_2 \tag{8.3}$$

```
#!/usr/bin/env python3

import sys
from z3 import *

"""
coordinates:
```

```
------------------------------
00 01 02 | 03 04 05 | 06 07 08
10 11 12 | 13 14 15 | 16 17 18
20 21 22 | 23 24 25 | 26 27 28
------------------------------
30 31 32 | 33 34 35 | 36 37 38
40 41 42 | 43 44 45 | 46 47 48
50 51 52 | 53 54 55 | 56 57 58
------------------------------
60 61 62 | 63 64 65 | 66 67 68
70 71 72 | 73 74 75 | 76 77 78
80 81 82 | 83 84 85 | 86 87 88
------------------------------
"""

s=Solver()

# using Python list comprehension, construct array of arrays of BitVec instances:
cells=[[BitVec('cell%d%d' % (r, c), 16) for c in range(9)] for r in range(9)]

# http://www.norvig.com/sudoku.html
# http://www.mirror.co.uk/news/weird-news/worlds-hardest-sudoku-can-you-242294
puzzle="
    ..53.....8......2..7..1.5..4....53...1..7...6..32...8..6.5....9..4....3......97.."

# process text line:
current_column=0
current_row=0
for i in puzzle:
        if i!='.':
                s.add(cells[current_row][current_column]==BitVecVal(int(i),16))
        current_column=current_column+1
        if current_column==9:
                current_column=0
                current_row=current_row+1

one=BitVecVal(1,16)
mask=BitVecVal(0b1111111110,16)

# for all 9 rows
for r in range(9):
        s.add(((one<<cells[r][0]) |
                (one<<cells[r][1]) |
                (one<<cells[r][2]) |
                (one<<cells[r][3]) |
                (one<<cells[r][4]) |
                (one<<cells[r][5]) |
                (one<<cells[r][6]) |
                (one<<cells[r][7]) |
                (one<<cells[r][8]))==mask)

# for all 9 columns
for c in range(9):
        s.add(((one<<cells[0][c]) |
                (one<<cells[1][c]) |
                (one<<cells[2][c]) |
                (one<<cells[3][c]) |
                (one<<cells[4][c]) |
                (one<<cells[5][c]) |
                (one<<cells[6][c]) |
                (one<<cells[7][c]) |
                (one<<cells[8][c]))==mask)

# enumerate all 9 squares
for r in range(0, 9, 3):
        for c in range(0, 9, 3):
                # add constraints for each 3*3 square:
                        s.add(one<<cells[r+0][c+0] |
                                one<<cells[r+0][c+1] |
                                one<<cells[r+0][c+2] |
                                one<<cells[r+1][c+0] |
                                one<<cells[r+1][c+1] |
                                one<<cells[r+1][c+2] |
                                one<<cells[r+2][c+0] |
                                one<<cells[r+2][c+1] |
                                one<<cells[r+2][c+2]==mask)

print (s.check())
#print (s.model())
m=s.model()

for r in range(9):
        for c in range(9):
                sys.stdout.write (str(m[cells[r][c]])+" ")
        print ("")
```

(https://smt.st/current_tree/puzzles/sudoku/1/sudoku_or_Z3.py)

Now it works much faster. Z3 handles OR operation over bit vectors better than addition?

```
% time python sudoku_or_Z3.py
1 4 5 3 2 7 6 9 8
8 3 9 6 5 4 1 2 7
6 7 2 9 1 8 5 4 3
4 9 6 1 8 5 3 7 2
2 1 8 4 7 3 9 5 6
7 5 3 2 9 6 4 8 1
3 6 7 5 4 2 8 1 9
9 8 4 7 6 1 2 3 5
5 2 1 8 3 9 7 6 4

real    0m1.429s
user    0m1.393s
sys     0m0.036s
```

The puzzle I used as example is dubbed as one of the hardest known [1] (well, for humans). It took ≈ 1.4 seconds on my Intel Core i3-3110M 2.4GHz notebook to solve it.

The second idea

My first approach is far from effective, I did what first came to my mind and worked. Another approach is to use `distinct` command from SMT-LIB, which tells Z3 that some variables must be distinct (or unique). This command is also available in Z3 Python interface.

I've rewritten my first Sudoku solver, now it operates over *Int sort*, it has `distinct` commands instead of bit operations, and now also other constraint added: each cell value must be in 1..9 range, because, otherwise, Z3 will offer (although correct) solution with too big and/or negative numbers.

```
#!/usr/bin/env python3

import sys
from z3 import *

"""
```

[1] http://www.mirror.co.uk/news/weird-news/worlds-hardest-sudoku-can-you-242294

```
------------------------------
00 01 02 | 03 04 05 | 06 07 08
10 11 12 | 13 14 15 | 16 17 18
20 21 22 | 23 24 25 | 26 27 28
------------------------------
30 31 32 | 33 34 35 | 36 37 38
40 41 42 | 43 44 45 | 46 47 48
50 51 52 | 53 54 55 | 56 57 58
------------------------------
60 61 62 | 63 64 65 | 66 67 68
70 71 72 | 73 74 75 | 76 77 78
80 81 82 | 83 84 85 | 86 87 88
------------------------------
"""

s=Solver()

# using Python list comprehension, construct array of arrays of BitVec instances:
cells=[[Int('cell%d%d' % (r, c)) for c in range(9)] for r in range(9)]

# http://www.norvig.com/sudoku.html
# http://www.mirror.co.uk/news/weird-news/worlds-hardest-sudoku-can-you-242294
puzzle="
   ..53.....8......2..7..1.5..4....53...1..7...6..32...8..6.5....9..4....3......97.."

# process text line:
current_column=0
current_row=0
for i in puzzle:
        if i!='.':
                s.add(cells[current_row][current_column]==int(i))
        current_column=current_column+1
        if current_column==9:
                current_column=0
                current_row=current_row+1

# this is important, because otherwise, Z3 will report correct solutions with too big
    and/or negative numbers in cells
for r in range(9):
        for c in range(9):
                s.add(cells[r][c]>=1)
                s.add(cells[r][c]<=9)

# for all 9 rows
for r in range(9):
        s.add(Distinct(cells[r][0],
                cells[r][1],
                cells[r][2],
                cells[r][3],
                cells[r][4],
                cells[r][5],
                cells[r][6],
                cells[r][7],
                cells[r][8]))

# for all 9 columns
for c in range(9):
        s.add(Distinct(cells[0][c],
                cells[1][c],
                cells[2][c],
                cells[3][c],
                cells[4][c],
                cells[5][c],
                cells[6][c],
                cells[7][c],
                cells[8][c]))

# enumerate all 9 squares
for r in range(0, 9, 3):
        for c in range(0, 9, 3):
                # add constraints for each 3*3 square:
                        s.add(Distinct(cells[r+0][c+0],
                                cells[r+0][c+1],
                                cells[r+0][c+2],
                                cells[r+1][c+0],
                                cells[r+1][c+1],
                                cells[r+1][c+2],
                                cells[r+2][c+0],
                                cells[r+2][c+1],
                                cells[r+2][c+2]))

print (s.check())
#print (s.model())
m=s.model()

for r in range(9):
        for c in range(9):
                sys.stdout.write (str(m[cells[r][c]])+" ")
        print ("")
```

(https://smt.st/current_tree/puzzles/sudoku/1/sudoku2_Z3.py)

```
% time python sudoku2_Z3.py
1 4 5 3 2 7 6 9 8
8 3 9 6 5 4 1 2 7
6 7 2 9 1 8 5 4 3
4 9 6 1 8 5 3 7 2
2 1 8 4 7 3 9 5 6
7 5 3 2 9 6 4 8 1
3 6 7 5 4 2 8 1 9
9 8 4 7 6 1 2 3 5
5 2 1 8 3 9 7 6 4

real    0m0.382s
user    0m0.346s
sys     0m0.036s
```

That's much faster.

Conclusion

SMT-solvers are so helpful, is that our Sudoku solver has nothing else, we have just defined relationships between variables (cells).

Homework

As it seems, true Sudoku puzzle is the one which has only one solution. The piece of code I've included here shows only the first one. Using the method described earlier (3.17, also called "model counting"), try to find more solutions, or prove that the solution you have just found is the only one possible.

Further reading

http://www.norvig.com/sudoku.html

Sudoku as a SAT problem

It’s also possible to represent Sudoku puzzle as a huge CNF equation and use SAT-solver to find solution, but it’s just trickier.

Some articles about it: *Building a Sudoku Solver with SAT*[2], Tjark Weber, *A SAT-based Sudoku Solver*[3], Ines Lynce, Joel Ouaknine, *Sudoku as a SAT Problem*[4], Gihwon Kwon, Himanshu Jain, *Optimized CNF Encoding for Sudoku Puzzles*[5].

SMT-solver can also use SAT-solver in its core, so it does all mundane translating work. As a “compiler”, it may not do this in the most efficient way, though.

8.1.2 Simple sudoku in SMT using functions and array

Surely, you may want to use functions and arrays, like you would do in any PL.

Arrays in SMT are immutable of course. You can’t modify them. (There is nothing to modify.) You can create a new array with a new element stored at some index, or replaced.

But here we will use only *select* operation on arrays.

Also, functions in SMT are more like macros.

```
; QF - no quantifiers used
; A - arrays used
; BV - bitvectors
(set-logic QF_ABV)
(set-info :smt-lib-version 2.0)

;-----------------------------
;00 01 02 | 03 04 05 | 06 07 08
;10 11 12 | 13 14 15 | 16 17 18
;20 21 22 | 23 24 25 | 26 27 28
;-----------------------------
;30 31 32 | 33 34 35 | 36 37 38
;40 41 42 | 43 44 45 | 46 47 48
;50 51 52 | 53 54 55 | 56 57 58
;-----------------------------
;60 61 62 | 63 64 65 | 66 67 68
;70 71 72 | 73 74 75 | 76 77 78
;80 81 82 | 83 84 85 | 86 87 88
;-----------------------------

; each array index is a 8-bit byte
; high 4-bit - column, low 4-bit - row

(declare-const cells (Array (_ BitVec 8) (_ BitVec 4)))

; input - byte. high 4-bit - column, low 4-bit - row
; output - result of AND
(define-fun limits_for_cell
        (
                (arg1 (_ BitVec 8))
        )
        Bool
                (and
                        (bvuge (select cells arg1) (_ bv1 4))
                        (bvule (select cells arg1) (_ bv9 4))
                )
)

; input - 4-bit, column
(define-fun limits_for_col
```

[2] https://dspace.mit.edu/bitstream/handle/1721.1/106923/6-005-fall-2011/contents/assignments/MIT6_005F11_ps4.pdf

[3] https://www.lri.fr/~conchon/mpri/weber.pdf

[4] http://sat.inesc-id.pt/~ines/publications/aimath06.pdf

[5] http://www.cs.cmu.edu/~hjain/papers/sudoku-as-SAT.pdf

```
        (
                (arg1 (_ BitVec 4)) ; input argument and its type
        )
        Bool ; output type
; enumerate all 9 rows in column:
                (and
                        (limits_for_cell (concat arg1 #x0))
                        (limits_for_cell (concat arg1 #x1))
                        (limits_for_cell (concat arg1 #x2))
                        (limits_for_cell (concat arg1 #x3))
                        (limits_for_cell (concat arg1 #x4))
                        (limits_for_cell (concat arg1 #x5))
                        (limits_for_cell (concat arg1 #x6))
                        (limits_for_cell (concat arg1 #x7))
                        (limits_for_cell (concat arg1 #x8))
                )
)

; enumerate all 9 columns in Sudoku
; 'connect' output to always-true
(assert (limits_for_col #x0))
(assert (limits_for_col #x1))
(assert (limits_for_col #x2))
(assert (limits_for_col #x3))
(assert (limits_for_col #x4))
(assert (limits_for_col #x5))
(assert (limits_for_col #x6))
(assert (limits_for_col #x7))
(assert (limits_for_col #x8))

; each column must have distinct values:
(define-fun distinct_col
        (
                (arg1 (_ BitVec 4)) ; input argument and its type
        )
        Bool ; output type
                (distinct
                        (select cells (concat arg1 #x0))
                        (select cells (concat arg1 #x1))
                        (select cells (concat arg1 #x2))
                        (select cells (concat arg1 #x3))
                        (select cells (concat arg1 #x4))
                        (select cells (concat arg1 #x5))
                        (select cells (concat arg1 #x6))
                        (select cells (concat arg1 #x7))
                        (select cells (concat arg1 #x8))
                )
)

; 'connect' output to always-true
(assert (distinct_col #x0))
(assert (distinct_col #x1))
(assert (distinct_col #x2))
(assert (distinct_col #x3))
(assert (distinct_col #x4))
(assert (distinct_col #x5))
(assert (distinct_col #x6))
(assert (distinct_col #x7))
(assert (distinct_col #x8))

; each row must have distinct values:
```

```
(define-fun distinct_row
        (
                (arg1 (_ BitVec 4)) ; input argument and its type
        )
        Bool ; output type
                (distinct
                        (select cells (concat #x0 arg1))
                        (select cells (concat #x1 arg1))
                        (select cells (concat #x2 arg1))
                        (select cells (concat #x3 arg1))
                        (select cells (concat #x4 arg1))
                        (select cells (concat #x5 arg1))
                        (select cells (concat #x6 arg1))
                        (select cells (concat #x7 arg1))
                        (select cells (concat #x8 arg1))
                )
)

; 'connect' output to always-true
(assert (distinct_row #x0))
(assert (distinct_row #x1))
(assert (distinct_row #x2))
(assert (distinct_row #x3))
(assert (distinct_row #x4))
(assert (distinct_row #x5))
(assert (distinct_row #x6))
(assert (distinct_row #x7))
(assert (distinct_row #x8))

; 9 values inside of 3*3 block must be distinct:
(define-fun distinct_block
        (
                (arg1 (_ BitVec 4)) ; two input arguments and their types
                (arg2 (_ BitVec 4))
        )
        Bool ; output type
                (distinct
                        (select cells (concat (bvadd arg1 #x0) (bvadd arg2 #x0)))
                        (select cells (concat (bvadd arg1 #x0) (bvadd arg2 #x1)))
                        (select cells (concat (bvadd arg1 #x0) (bvadd arg2 #x2)))
                        (select cells (concat (bvadd arg1 #x1) (bvadd arg2 #x0)))
                        (select cells (concat (bvadd arg1 #x1) (bvadd arg2 #x1)))
                        (select cells (concat (bvadd arg1 #x1) (bvadd arg2 #x2)))
                        (select cells (concat (bvadd arg1 #x2) (bvadd arg2 #x0)))
                        (select cells (concat (bvadd arg1 #x2) (bvadd arg2 #x1)))
                        (select cells (concat (bvadd arg1 #x2) (bvadd arg2 #x2)))
                )
)

; enumerate all 9 3*3 blocks.
; 'connect' output to always-true
(assert (distinct_block #x0 #x0))
(assert (distinct_block #x0 #x3))
(assert (distinct_block #x0 #x6))
(assert (distinct_block #x3 #x0))
(assert (distinct_block #x3 #x3))
(assert (distinct_block #x3 #x6))
(assert (distinct_block #x6 #x0))
(assert (distinct_block #x6 #x3))
(assert (distinct_block #x6 #x6))
```

```
; http://www.norvig.com/sudoku.html
; http://www.mirror.co.uk/news/weird-news/worlds-hardest-sudoku-can-you-242294

;..53..... - 0
;8......2. - 1
;.7..1.5.. - 2
;4....53.. - 3
;.1..7...6 - 4
;..32...8. - 5
;.6.5....9 - 6
;..4....3. - 7
;.....97.. - 8

; set input:
(assert (= (select cells #x02) #x5))
(assert (= (select cells #x03) #x3))
(assert (= (select cells #x10) #x8))
(assert (= (select cells #x17) #x2))
(assert (= (select cells #x21) #x7))
(assert (= (select cells #x24) #x1))
(assert (= (select cells #x26) #x5))
(assert (= (select cells #x30) #x4))
(assert (= (select cells #x35) #x5))
(assert (= (select cells #x36) #x3))
(assert (= (select cells #x41) #x1))
(assert (= (select cells #x44) #x7))
(assert (= (select cells #x48) #x6))
(assert (= (select cells #x52) #x3))
(assert (= (select cells #x53) #x2))
(assert (= (select cells #x57) #x8))
(assert (= (select cells #x61) #x6))
(assert (= (select cells #x63) #x5))
(assert (= (select cells #x68) #x9))
(assert (= (select cells #x72) #x4))
(assert (= (select cells #x77) #x3))
(assert (= (select cells #x84) #x9))
(assert (= (select cells #x86) #x7))

(check-sat)
(get-model)
```

(https://smt.st/current_tree/puzzles/sudoku/2/sudoku_array.smt)

Let's see how different SMT solvers produce results, dumping array's content:

Listing 8.1: STP

```
(model
( define-fun |cells|  (_ BitVec 8) (_ BitVec 4) #x70 #x5 )
( define-fun |cells|  (_ BitVec 8) (_ BitVec 4) #x62 #x1 )
( define-fun |cells|  (_ BitVec 8) (_ BitVec 4) #x38 #x2 )
( define-fun |cells|  (_ BitVec 8) (_ BitVec 4) #x28 #x3 )
( define-fun |cells|  (_ BitVec 8) (_ BitVec 4) #x85 #x4 )

...

( define-fun |cells|  (_ BitVec 8) (_ BitVec 4) #x40 #x9 )
( define-fun |cells|  (_ BitVec 8) (_ BitVec 4) #x84 #x9 )
( define-fun |cells|  (_ BitVec 8) (_ BitVec 4) #x66 #x8 )
( define-fun |cells|  (_ BitVec 8) (_ BitVec 4) #x21 #x7 )
( define-fun |cells|  (_ BitVec 8) (_ BitVec 4) #x63 #x5 )
)
```

Some SMT solvers dumps a function that create an array using *store* operations:

Listing 8.2: Z3

```
(
  (define-fun cells () (Array (_ BitVec 8) (_ BitVec 4))
    (let ((a!1 (store (store (store ((as const (Array (_ BitVec 8) (_ BitVec 4)))
                                  #x1)
                                #x07
                                #x7)
                         #x18
                         #x3)
                  #x71
                  #x9)))
(let ((a!2 (store (store (store (store a!1 #x38 #x2) #x45 #x3) #x60 #x7)
                  #x65
                  #x8)))
(let ((a!3 (store (store (store (store a!2 #x50 #x6) #x43 #x8) #x12 #x9)
                  #x28
                  #x8)))

...

(let ((a!18 (store (store (store (store a!17 #x21 #x7) #x17 #x2) #x10 #x8)
                   #x03
                   #x3)))
  (store a!18 #x02 #x5))))))))))))))))))))
)
```

Some other SMT solvers create an ITE-tree, that will behave exactly like the array filled with correct values. In fact, you can copy and paste this output and use it as a function in your SMT code:

Listing 8.3: Boolector

```
(
  (define-fun cells (
   (cells_x0 (_ BitVec 8))) (_ BitVec 4)
    (ite (= cells_x0 (_ bv8 8)) (_ bv4 4)
    (ite (= cells_x0 (_ bv7 8)) (_ bv7 4)
    (ite (= cells_x0 (_ bv6 8)) (_ bv6 4)
    (ite (= cells_x0 (_ bv5 8)) (_ bv9 4)
    (ite (= cells_x0 (_ bv4 8)) (_ bv8 4)
    (ite (= cells_x0 (_ bv3 8)) (_ bv3 4)

...

    (ite (= cells_x0 (_ bv131 8)) (_ bv1 4)
    (ite (= cells_x0 (_ bv130 8)) (_ bv8 4)
    (ite (= cells_x0 (_ bv128 8)) (_ bv2 4)
    (ite (= cells_x0 (_ bv129 8)) (_ bv3 4)
      (_ bv0 4)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
         ))))))))))))
)
```

Listing 8.4: Bitwuzla

```
(
  (define-fun cells ((@x0 (_ BitVec 8))) (_ BitVec 4)
    (ite (= @x0 (_ bv8 8)) (_ bv8 4)
    (ite (= @x0 (_ bv7 8)) (_ bv7 4)
    (ite (= @x0 (_ bv6 8)) (_ bv6 4)
    (ite (= @x0 (_ bv5 8)) (_ bv9 4)
    (ite (= @x0 (_ bv4 8)) (_ bv2 4)
    (ite (= @x0 (_ bv3 8)) (_ bv3 4)
...
```

```
    (ite (= @x0 (_ bv131 8)) (_ bv1 4)
    (ite (= @x0 (_ bv130 8)) (_ bv8 4)
    (ite (= @x0 (_ bv128 8)) (_ bv3 4)
    (ite (= @x0 (_ bv129 8)) (_ bv2 4)
      (_ bv0 4))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
          ))))))))))))
)
```

Unlike Z3, CVC5 dumps a function that constructs array without using *let*, but again, using nested *store* operations:

Listing 8.5: CVC5

```
(
(define-fun cells () (Array (_ BitVec 8) (_ BitVec 4)) (store (store (store (store (
   store (store (store (store (store (store (store (store (store (store (store (store
    (store (store (store (store (store (store (store (store (store (store (store (
   store (store (store (store (store (store (store (store (store (store (store (store
    (store (store (store (store (store (store (store (store (store (store (store (
   store (store (store (store (store (store (store (store (store (store (store (store
    (store (store (store (store (store (store (store (store (store (store (store (
   store (store (store (store (store (store (store (store ((as const (Array (_ BitVec
    8) (_ BitVec 4))) #b0000) #b00000010 #b0101) #b00000011 #b0011) #b00010000 #b1000
   ) #b00010111 #b0010) #b00100001 #b0111) #b00100100 #b0001) #b00100110 #b0101) #
   b00110000 #b0100) #b00110101 #b0101) #b00110110 #b0011) #b01000001 #b0001) #
   b01000100 #b0111) #b01001000 #b0110) #b01010010 #b0011) #b01010011 #b0010) #
   b01010111 #b1000) #b01100001 #b0110) #b01100011 #b0101) #b01101000 #b1001) #
   b01110010 #b0100) #b01110111 #b0011) #b10000100 #b1001) #b10000110 #b0111) #
   b00000000 #b0001) #b00000001 #b0100) #b00000100 #b0010) #b00000101 #b1001) #
   b00000110 #b0110) #b00000111 #b0111) #b00001000 #b1000) #b00010001 #b0011) #
   b00010010 #b1001) #b00010011 #b0110) #b00010100 #b0101) #b00010101 #b0111) #
   b00010110 #b0001) #b00011000 #b0100) #b00100000 #b0010) #b00100010 #b0110) #
   b00100011 #b0100) #b00100101 #b1000) #b00100111 #b1001) #b00101000 #b0011) #
   b00110001 #b1000) #b00110010 #b0111) #b00110011 #b1001) #b00110100 #b0110) #
   b00110111 #b0001) #b00111000 #b0010) #b01000000 #b1001) #b01000010 #b0010) #
   b01000011 #b1000) #b01000101 #b0011) #b01000110 #b0100) #b01000111 #b0101) #
   b01010000 #b0110) #b01010001 #b0101) #b01010100 #b0100) #b01010101 #b0001) #
   b01010110 #b1001) #b01011000 #b0111) #b01100000 #b0111) #b01100010 #b0001) #
   b01100100 #b0011) #b01100101 #b0010) #b01100110 #b1000) #b01100111 #b0100) #
   b01110000 #b0101) #b01110001 #b1001) #b01110011 #b0111) #b01110100 #b1000) #
   b01110101 #b0110) #b01110110 #b0010) #b01111000 #b0001) #b10000000 #b0011) #
   b10000001 #b0010) #b10000010 #b1000) #b10000011 #b0001) #b10000101 #b0100) #
   b10000111 #b0110) #b10001000 #b0101))
)
```

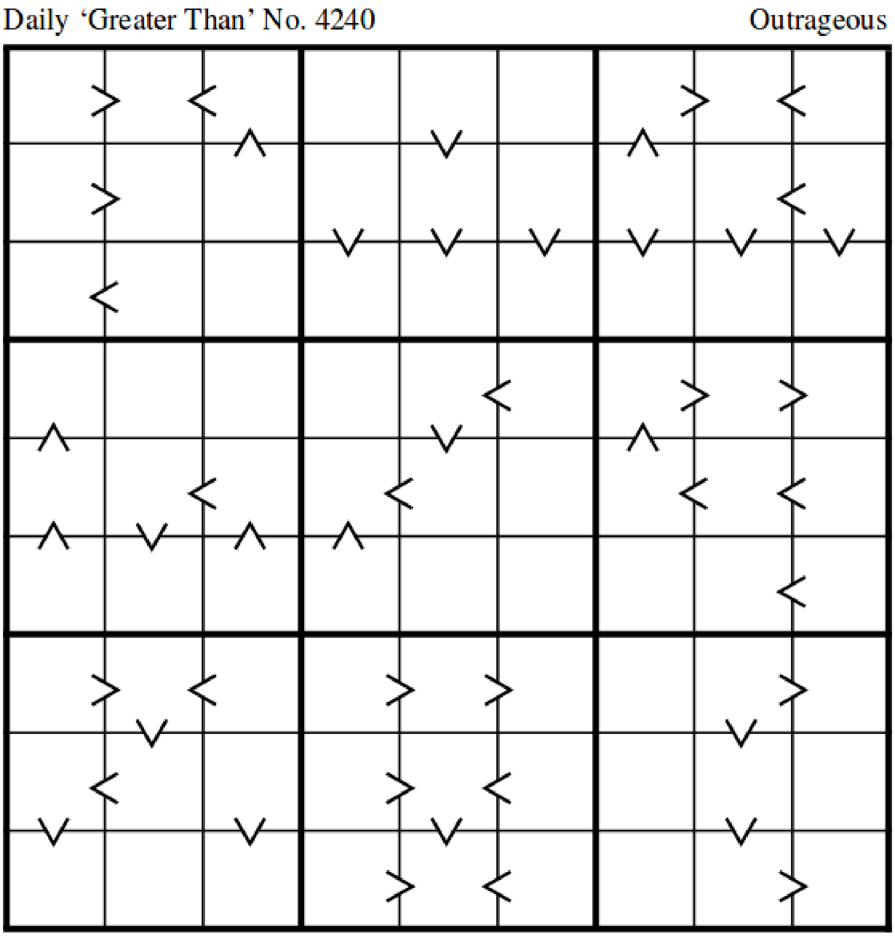
Yices does something slightly different:

Listing 8.6: Yices

```
(= cells @fun_172)
(function @fun_172
 (type (Array (_ BitVec 8) (_ BitVec 4)))
 (= (@fun_172 #b00000011) #b0011)
 (= (@fun_172 #b00010000) #b1000)
 (= (@fun_172 #b00010111) #b0010)
 (= (@fun_172 #b00100001) #b0111)
 (= (@fun_172 #b00100100) #b0001)

...

 (= (@fun_172 #b10000101) #b0100)
 (= (@fun_172 #b10000111) #b0110)
 (default #b0101))
```

8.1.3 Greater Than Sudoku

I've found this on http://www.killersudokuonline.com:



Copyright (c) 2016, killersudokuonline.com

Figure 8.1

It can be solved easily with Z3. I've took the same piece of code I used for the usual Sudoku: 8.1.1.
... and added this:

```
...

"""
Subsquares:

------------------------------
         |          |
 1,1     | 1,2      | 1,3
         |          |
------------------------------
         |          |
 2,1     | 2,2      | 2,3
         |          |
------------------------------
         |          |
 3,1     | 3,2      | 3,3
         |          |
------------------------------
"""
```

```
# from http://www.killersudokuonline.com/puzzles/2017/puzzle-GD4hzi164344.pdf

# subsquare 1,1:
s.add(cells[0][0]>cells[0][1])
s.add(cells[1][0]>cells[1][1])
s.add(cells[2][0]<cells[2][1])

s.add(cells[0][1]<cells[0][2])
s.add(cells[0][2]<cells[1][2])

# subsquare 1,2:
s.add(cells[0][4]>cells[1][4])
s.add(cells[1][3]>cells[2][3])
s.add(cells[1][4]>cells[2][4])
s.add(cells[1][5]>cells[2][5])

# subsquare 1,3:
s.add(cells[0][6]>cells[0][7])
s.add(cells[0][7]<cells[0][8])
s.add(cells[0][6]<cells[1][6])
s.add(cells[1][7]<cells[1][8])
s.add(cells[1][6]>cells[2][6])
s.add(cells[1][7]>cells[2][7])
s.add(cells[1][8]>cells[2][8])

# subsquare 2,1:
s.add(cells[3][0]<cells[4][0])
s.add(cells[4][0]<cells[5][0])
s.add(cells[4][1]<cells[4][2])
s.add(cells[4][0]<cells[5][0])
s.add(cells[4][1]>cells[5][1])
s.add(cells[4][2]<cells[5][2])

# subsquare 2,2:
s.add(cells[3][4]>cells[4][4])
s.add(cells[3][4]<cells[3][5])
s.add(cells[4][3]<cells[4][4])
s.add(cells[4][3]<cells[5][3])

# subsquare 2,3:
s.add(cells[3][6]>cells[3][7])
s.add(cells[3][7]>cells[3][8])
s.add(cells[3][6]>cells[4][6])
s.add(cells[4][6]<cells[4][7])
s.add(cells[4][7]<cells[4][8])
s.add(cells[5][7]<cells[5][8])

# subsquare 3,1:
s.add(cells[6][0]>cells[6][1])
s.add(cells[6][1]<cells[6][2])
s.add(cells[6][1]>cells[7][1])
s.add(cells[7][0]<cells[7][1])
s.add(cells[7][0]>cells[8][0])
s.add(cells[7][2]>cells[8][2])

# subsquare 3,2:
s.add(cells[6][3]>cells[6][4])
s.add(cells[6][4]>cells[6][5])
s.add(cells[7][3]>cells[7][4])
s.add(cells[7][4]<cells[7][5])
```

```
s.add(cells[8][3]>cells[8][4])
s.add(cells[8][4]<cells[8][5])
s.add(cells[7][4]>cells[8][4])

# subsquare 3,3:
s.add(cells[6][7]>cells[6][8])
s.add(cells[6][7]>cells[7][7])
s.add(cells[7][7]>cells[8][7])
s.add(cells[8][7]>cells[8][8])

...
```

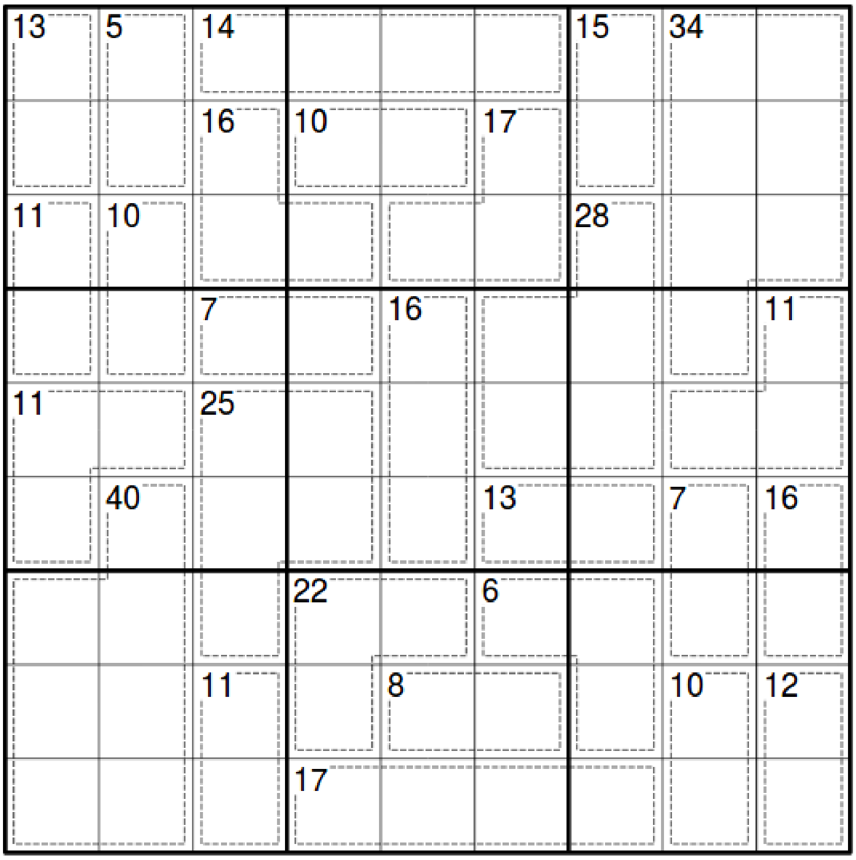
(The full file: https://smt.st/current_tree/puzzles/sudoku/GT/sudoku_GT.py)

The puzzle marked as “Outrageous” (for humans?), however it took ≈ 30 seconds on my old Intel Xeon E3-1220 3.10GHz to solve it:

```
7 3 4 6 9 2 5 1 8
2 1 5 8 3 7 9 4 6
6 8 9 5 1 4 7 2 3
1 7 3 2 8 9 6 5 4
5 4 6 1 7 3 2 8 9
9 2 8 4 5 6 1 3 7
8 6 7 3 2 1 4 9 5
4 5 2 9 6 8 3 7 1
3 9 1 7 4 5 8 6 2
```

8.1.4 Solving Killer Sudoku

I’ve found this on https://krazydad.com/killersudoku/sfiles/KD_Killer_ST16_8_v52.pdf:



© 2017 KrazyDad.com

Figure 8.2

There are "cages", each cage must have distinct digits, and its sum must be equal to the number written there in a manner of crossword. See also: https://en.wikipedia.org/wiki/Killer_sudoku.

This is also piece of cake for Z3. I've took the same piece of code I used for usual Sudoku (8.1.1).

```
...

cage=[cells[0][0], cells[1][0]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==13)

cage=[cells[0][1], cells[1][1]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==5)

cage=[cells[0][2], cells[0][3], cells[0][4], cells[0][5]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==14)

cage=[cells[0][6], cells[1][6]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==15)

cage=[cells[0][7], cells[0][8], cells[1][7], cells[1][8], cells[2][7], cells[2][8],
    cells[3][7]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==34)

cage=[cells[1][2], cells[2][2], cells[2][3]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==16)

cage=[cells[1][3], cells[1][4]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==10)

cage=[cells[1][5], cells[2][4], cells[2][5]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==17)

cage=[cells[2][6], cells[3][5], cells[3][6], cells[4][5], cells[4][6]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==28)

cage=[cells[3][2], cells[3][3]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==7)

cage=[cells[3][4], cells[4][4], cells[5][4]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==16)

cage=[cells[3][8], cells[4][7], cells[4][8]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==11)

cage=[cells[4][0], cells[4][1], cells[5][0]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==11)

cage=[cells[4][2], cells[4][3], cells[5][2], cells[5][3], cells[6][2]]
s.add(Distinct(*cage))
```

```
s.add(Sum(*cage)==25)

cage=[cells[5][1], cells[6][0], cells[6][1], cells[7][0], cells[7][1], cells[8][0],
    cells[8][1]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==40)

cage=[cells[5][5], cells[5][6]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==13)

cage=[cells[5][7], cells[6][7]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==7)

cage=[cells[5][8], cells[6][8]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==16)

cage=[cells[6][3], cells[6][4], cells[7][3]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==22)

cage=[cells[6][5], cells[6][6], cells[7][6]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==6)

cage=[cells[7][2], cells[8][2]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==11)

cage=[cells[7][4], cells[7][5]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==8)

cage=[cells[7][7], cells[8][7]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==10)

cage=[cells[7][8], cells[8][8]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==12)

cage=[cells[8][3], cells[8][4], cells[8][5], cells[8][6]]
s.add(Distinct(*cage))
s.add(Sum(*cage)==17)

...
```

(The full file: https://smt.st/current_tree/puzzles/sudoku/killer/killer_sudoku.py)

The puzzle marked as “Super-Tough Killer Sudoku Puzzle” (again, for humans?), however it took ≈ 30 seconds on my old Intel Xeon E3-1220 3.10GHz to solve it:

```
5 3 4 7 1 2 8 9 6
8 2 1 4 6 9 7 5 3
9 6 7 8 3 5 4 2 1
2 4 6 1 9 7 3 8 5
7 1 9 3 5 8 6 4 2
3 8 5 6 2 4 9 1 7
4 7 2 5 8 3 1 6 9
6 5 8 9 7 1 2 3 4
1 9 3 2 4 6 5 7 8
```

8.1.5 KLEE

(All timings are when KLEE 3.0 running on AMD Ryzen 5 3600.)

I've also rewritten Sudoku example (8.1.1) for KLEE:

```
#include <stdint.h>
#include <assert.h>

#include "klee.h"

/*
coordinates:
------------------------------
00 01 02 | 03 04 05 | 06 07 08
10 11 12 | 13 14 15 | 16 17 18
20 21 22 | 23 24 25 | 26 27 28
------------------------------
30 31 32 | 33 34 35 | 36 37 38
40 41 42 | 43 44 45 | 46 47 48
50 51 52 | 53 54 55 | 56 57 58
------------------------------
60 61 62 | 63 64 65 | 66 67 68
70 71 72 | 73 74 75 | 76 77 78
80 81 82 | 83 84 85 | 86 87 88
------------------------------
*/

uint8_t cells[9][9];

// http://www.norvig.com/sudoku.html
// http://www.mirror.co.uk/news/weird-news/worlds-hardest-sudoku-can-you-242294
char *puzzle
    ="..53.....8......2..7..1.5..4....53...1..7...6..32...8..6.5....9..4....3......97..";


int main()
{
        klee_make_symbolic(cells, sizeof cells, "cells");

        // process text line:
        for (int row=0; row<9; row++)
                for (int column=0; column<9; column++)
                {
                        char c=puzzle[row*9 + column];
                        if (c!='.')
                        {
                                if (cells[row][column]!=c-'0') return 0;
                        }
                        else
                        {
                                // limit cells values to 1..9:
                                if (cells[row][column]<1) return 0;
                                if (cells[row][column]>9) return 0;
                        };
                };

        // for all 9 rows
        for (int row=0; row<9; row++)
        {

                if (((1<<cells[row][0]) |
                        (1<<cells[row][1]) |
                        (1<<cells[row][2]) |
                        (1<<cells[row][3]) |
                        (1<<cells[row][4]) |
                        (1<<cells[row][5]) |
                        (1<<cells[row][6]) |
                        (1<<cells[row][7]) |
                        (1<<cells[row][8]))!=0x3FE ) return 0; // 11 1111 1110
        };

        // for all 9 columns
        for (int c=0; c<9; c++)
        {
                if (((1<<cells[0][c]) |
                        (1<<cells[1][c]) |
                        (1<<cells[2][c]) |
                        (1<<cells[3][c]) |
                        (1<<cells[4][c]) |
                        (1<<cells[5][c]) |
                        (1<<cells[6][c]) |
                        (1<<cells[7][c]) |
                        (1<<cells[8][c]))!=0x3FE ) return 0; // 11 1111 1110
        };

        // enumerate all 9 squares
        for (int r=0; r<9; r+=3)
                for (int c=0; c<9; c+=3)
                {
                        // add constraints for each 3*3 square:
                        if ((1<<cells[r+0][c+0] |
                                1<<cells[r+0][c+1] |
                                1<<cells[r+0][c+2] |
                                1<<cells[r+1][c+0] |
                                1<<cells[r+1][c+1] |
                                1<<cells[r+1][c+2] |
                                1<<cells[r+2][c+0] |
                                1<<cells[r+2][c+1] |
                                1<<cells[r+2][c+2])!=0x3FE ) return 0; // 11 1111 1110
                };

        // at this point, all constraints must be satisfied
        assert(0);
};
```

Let's run it:

```
% clang -emit-llvm -c -g -O0 -Xclang -disable-O0-optnone -I /tmp/klee_src/include/klee/ klee_sudoku_or1.c

...

% time klee --libc=uclibc --optimize klee_sudoku_or1.bc
...
KLEE: ERROR: klee_sudoku_or1.c:96: ASSERTION FAIL: 0
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 23675
KLEE: done: completed paths = 102
KLEE: done: partially completed paths = 1
```

```
KLEE: done: generated tests = 103

real    6m4.584s
user    6m3.999s
```

Now this is really slower (on AMD Ryzen 5 3600) in comparison to Z3Py solution (8.1.1).

But the answer is correct:

```
% ls klee-last/*err
klee-last/test000103.external.err

% ktest-tool klee-last/test000103.ktest
ktest file : 'klee-last/test000103.ktest'
args       : ['klee_sudoku_or1.bc']
num objects: 1
object 0: name: 'cells'
object 0: size: 81
object 0: data: b'\x01\x04\x05\x03\x02\x07\x06\t\x08\x08\x03\t\x06\x05\x04\x01\x02\
    x07\x06\x07\x02\t\x01\x08\x05\x04\x03\x04\t\x06\x01\x08\x05\x03\x07\x02\x02\x01\
    x08\x04\x07\x03\t\x05\x06\x07\x05\x03\x02\t\x06\x04\x08\x01\x03\x06\x07\x05\x04\
    x02\x08\x01\t\t\x08\x04\x07\x06\x01\x02\x03\x05\x05\x02\x01\x08\x03\t\x07\x06\x04'
object 0: hex : 0
    x0104050302070609080803090605040102070607020901080504030409060108050307020201080407030905

object 0: text:
    .................................................................................
```

Character `\t` has code of 9 in C/C++, and KLEE prints byte array as a C/C++ string, so it shows some values in such way. We can just keep in mind that there is 9 at the each place where we see `\t`. The solution, while not properly formatted, correct indeed.

By the way, at lines 45 and 46 you may see how we tell to KLEE that all array elements must be within some limits. If we comment these lines out, we've got this:

```
...
KLEE: ERROR: klee_sudoku_or1.c:54: overshift error
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: klee_sudoku_or1.c:55: overshift error
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: klee_sudoku_or1.c:58: overshift error
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: klee_sudoku_or1.c:59: overshift error
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: klee_sudoku_or1.c:60: overshift error
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: klee_sudoku_or1.c:61: overshift error
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: klee_sudoku_or1.c:62: overshift error
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: klee_sudoku_or1.c:56: overshift error
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: klee_sudoku_or1.c:57: overshift error
KLEE: NOTE: now ignoring this error at this location
...
```

KLEE warns us that shift value at lines 54...62 is too big. Indeed, KLEE may try all byte values up to 255 (0xFF), which are pointless to use there, and may be a symptom of error or bug, so KLEE warns about it.

Now let's use `klee_assume()` again:

```
#include <stdint.h>
#include <assert.h>

#include "klee.h"

```

```
/*
coordinates:
------------------------------
00 01 02 | 03 04 05 | 06 07 08
10 11 12 | 13 14 15 | 16 17 18
20 21 22 | 23 24 25 | 26 27 28
------------------------------
30 31 32 | 33 34 35 | 36 37 38
40 41 42 | 43 44 45 | 46 47 48
50 51 52 | 53 54 55 | 56 57 58
------------------------------
60 61 62 | 63 64 65 | 66 67 68
70 71 72 | 73 74 75 | 76 77 78
80 81 82 | 83 84 85 | 86 87 88
------------------------------
*/

uint8_t cells[9][9];

// http://www.norvig.com/sudoku.html
// http://www.mirror.co.uk/news/weird-news/worlds-hardest-sudoku-can-you-242294
char *puzzle
   ="..53.....8......2..7..1.5..4....53...1..7...6..32...8..6.5....9..4....3......97..";


int main()
{
        klee_make_symbolic(cells, sizeof cells, "cells");

        // process text line:
        for (int row=0; row<9; row++)
                for (int column=0; column<9; column++)
                {
                        char c=puzzle[row*9 + column];
                        if (c!='.')
                                klee_assume (cells[row][column]==c-'0');
                        else
                        {
                                klee_assume (cells[row][column]>=1);
                                klee_assume (cells[row][column]<=9);
                        };
                };


        // for all 9 rows
        for (int row=0; row<9; row++)
        {

                klee_assume (((1<<cells[row][0]) |
                                (1<<cells[row][1]) |
                                (1<<cells[row][2]) |
                                (1<<cells[row][3]) |
                                (1<<cells[row][4]) |
                                (1<<cells[row][5]) |
                                (1<<cells[row][6]) |
                                (1<<cells[row][7]) |
                                (1<<cells[row][8]))==0x3FE ); // 11 1111 1110

        };

        // for all 9 columns
        for (int c=0; c<9; c++)
        {
                klee_assume (((1<<cells[0][c]) |
                                (1<<cells[1][c]) |
                                (1<<cells[2][c]) |
                                (1<<cells[3][c]) |
                                (1<<cells[4][c]) |
                                (1<<cells[5][c]) |
                                (1<<cells[6][c]) |
                                (1<<cells[7][c]) |
                                (1<<cells[8][c]))==0x3FE ); // 11 1111 1110
        };


        // enumerate all 9 squares
        for (int r=0; r<9; r+=3)
                for (int c=0; c<9; c+=3)
                {
                        // add constraints for each 3*3 square:
                        klee_assume ((1<<cells[r+0][c+0] |
                                        1<<cells[r+0][c+1] |
                                        1<<cells[r+0][c+2] |
                                        1<<cells[r+1][c+0] |
                                        1<<cells[r+1][c+1] |
                                        1<<cells[r+1][c+2] |
                                        1<<cells[r+2][c+0] |
                                        1<<cells[r+2][c+1] |
                                        1<<cells[r+2][c+2])==0x3FE ); // 11 1111 1110
                };

        // at this point, all constraints must be satisfied
        assert(0);
};
```

```
...
KLEE: ERROR: klee_sudoku_or2.c:96: ASSERTION FAIL: 0
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 16739
KLEE: done: completed paths = 0
KLEE: done: partially completed paths = 1
KLEE: done: generated tests = 1

real    0m18.663s
user    0m18.409s
...
```

That works much faster: because only one path to explore. And, as we see, the only one path has been found (one we actually interesting in it) instead of 161.

It's still much slower than Z3Py solution, though.

8.1.6 Sudoku in SAT

One might think that we can encode each 1..9 number in binary form: 5 bits or variables would be enough. But there is even simpler way: allocate 9 bits, where only one bit will be *True*. The number 1 can be encoded as [1, 0, 0, 0, 0, 0, 0, 0, 0], the number 3 as [0, 0, 1, 0, 0, 0, 0, 0, 0], etc. Seems uneconomical? Yes, but other operations would be simpler.

First of all, we'll reuse important POPCNT1 function I've described earlier: 8.4.1.

The second important operation we need to invent is making 9 numbers unique. If each number is encoded as 9-bits vector, 9 numbers can form a matrix, like:

```
0 0 0 0 0 0 1 0 0 <- 1st number
0 0 0 0 0 1 0 0 0 <- 2nd number
0 1 0 0 0 0 0 0 0 <- ...
0 0 1 0 0 0 0 0 0 <- ...
0 0 0 0 0 0 0 0 1 <- ...
0 0 0 0 1 0 0 0 0 <- ...
0 0 0 0 0 0 0 1 0 <- ...
1 0 0 0 0 0 0 0 0 <- ...
0 0 0 1 0 0 0 0 0 <- 9th number
```

Now we will use a POPCNT1 function to make each row in the matrix to contain only one *True* bit, that will preserve consistency in encoding, since no vector can contain more than 1 *True* bit, or no *True* bits at all. Then we will use a POPCNT1 function again to make all columns in the matrix to have only one single *True* bit. That will make all rows in matrix unique, in other words, all 9 encoded numbers will always be unique.

After applying POPCNT1 function 9+9=18 times we'll have 9 unique numbers in 1..9 range.

Using that operation we can make each row of Sudoku puzzle unique, each column unique and also each $3 \cdot 3 = 9$ box.

```
#!/usr/bin/env python3
#-*- coding: utf-8 -*-

import itertools, subprocess, os

# global variables:
clauses=[]
vector_names={}
last_var=1

BITS_PER_VECTOR=9

def read_lines_from_file (fname):
    f=open(fname)
    new_ar=[item.rstrip() for item in f.readlines()]
    f.close()
    return new_ar

def run_minisat (CNF_fname):
    child = subprocess.Popen(["minisat", CNF_fname, "results.txt"], stdout=subprocess
        .PIPE)
    child.wait()
    # 10 is SAT, 20 is UNSAT
    if child.returncode==20:
        os.remove ("results.txt")
        return None

    if child.returncode!=10:
        print ("(minisat) unknown retcode: ", child.returncode)
        exit(0)

    solution=read_lines_from_file("results.txt")[1].split(" ")
    os.remove ("results.txt")

    return solution

def write_CNF(fname, clauses, VARS_TOTAL):
    f=open(fname, "w")
    f.write ("p cnf "+str(VARS_TOTAL)+" "+str(len(clauses))+"\n")
    [f.write(" ".join(c)+" 0\n") for c in clauses]
    f.close()

def neg(v):
    return "-"+v
```

```
def add_popcnt1(vars):
    global clauses
    # enumerate all possible pairs
    # no pair can have both True's
    # so add "~var OR ~var2"
    for pair in itertools.combinations(vars, r=2):
        clauses.append([neg(pair[0]), neg(pair[1])])
    # at least one var must be present:
    clauses.append(vars)

def make_distinct_bits_in_vector(vec_name):
    global vector_names
    global last_var

    add_popcnt1([vector_names[(vec_name,i)] for i in range(BITS_PER_VECTOR)])

def make_distinct_vectors(vectors):
    # take each bit from all vectors, call add_popcnt1()
    for i in range(BITS_PER_VECTOR):
        add_popcnt1([vector_names[(vec,i)] for vec in vectors])

def cvt_vector_to_number(vec_name, solution):
    for i in range(BITS_PER_VECTOR):
        if vector_names[(vec_name,i)] in solution:
            # variable present in solution as non-negated (without a "-" prefix)
            return i+1
    raise AssertionError

def alloc_var():
    global last_var
    last_var=last_var+1
    return str(last_var-1)

def alloc_vector(l, name):
    global last_var
    global vector_names
    rt=[]
    for i in range(l):
        v=alloc_var()
        vector_names[(name,i)]=v
        rt.append(v)
    return rt

def add_constant(var,b):
    global clauses
    if b==True or b==1:
        clauses.append([var])
    else:
        clauses.append([neg(var)])

# vec is a list of True/False/0/1
def add_constant_vector(vec_name, vec):
    global vector_names
    for i in range(BITS_PER_VECTOR):
        add_constant (vector_names[(vec_name, i)], vec[i])

# 1 -> [1, 0, 0, 0, 0, 0, 0, 0, 0]
# 3 -> [0, 0, 1, 0, 0, 0, 0, 0, 0]
def number_to_vector(n):
    rt=[0]*(n-1)
    rt.append(1)
    rt=rt+[0]*(BITS_PER_VECTOR-len(rt))
    return rt

"""
coordinates we're using here:

+--------+--------+--------+
|11 12 13|14 15 16|17 18 19|
|21 22 23|24 25 26|27 28 29|
|31 32 33|34 35 36|37 38 39|
+--------+--------+--------+
|41 42 43|44 45 46|47 48 49|
|51 52 53|54 55 56|57 58 59|
|61 62 63|64 65 66|67 68 69|
+--------+--------+--------+
|71 72 73|74 75 76|77 78 79|
|81 82 83|84 85 86|87 88 89|
|91 92 93|94 95 96|97 98 99|
+--------+--------+--------+
"""
def make_vec_name(row, col):
    return "cell"+str(row)+str(col)

def puzzle_to_clauses (puzzle):
    # process text line:
    current_column=1
    current_row=1
    for i in puzzle:
        if i!='.':
            add_constant_vector(make_vec_name(current_row, current_column),
               number_to_vector(int(i)))
        current_column=current_column+1
        if current_column==10:
            current_column=1
            current_row=current_row+1

def print_solution(solution):
    for row in range(1,9+1):
        # print row:
        print (" ".join([str(cvt_vector_to_number(make_vec_name(row, col), solution))
            for col in range(1,9+1)]))

def main():
    # allocate 9*9*9=729 variables:
    for row in range(1, 9+1):
        for col in range(1, 9+1):
            alloc_vector(9, make_vec_name(row, col))
            make_distinct_bits_in_vector(make_vec_name(row, col))

    # variables in each row are unique:
    for row in range(1, 9+1):
        make_distinct_vectors([make_vec_name(row, col) for col in range(1, 9+1)])

    # variables in each column are unique:
    for col in range(1, 9+1):
        make_distinct_vectors([make_vec_name(row, col) for row in range(1, 9+1)])

    # variables in each 3*3 box are unique:
    for row in range(1, 9+1, 3):
        for col in range(1, 9+1, 3):
            tmp=[]
            tmp.append(make_vec_name(row+0, col+0))
            tmp.append(make_vec_name(row+0, col+1))
            tmp.append(make_vec_name(row+0, col+2))
            tmp.append(make_vec_name(row+1, col+0))
            tmp.append(make_vec_name(row+1, col+1))
            tmp.append(make_vec_name(row+1, col+2))
            tmp.append(make_vec_name(row+2, col+0))
            tmp.append(make_vec_name(row+2, col+1))
            tmp.append(make_vec_name(row+2, col+2))
            make_distinct_vectors(tmp)

    # http://www.norvig.com/sudoku.html
    # http://www.mirror.co.uk/news/weird-news/worlds-hardest-sudoku-can-you-242294
    puzzle_to_clauses("
        ..53.....8......2..7..1.5..4....53...1..7...6..32...8..6.5....9..4....3......97..
        ")

    print ("len(clauses)=",len(clauses))
    write_CNF("1.cnf", clauses, last_var-1)
    solution=run_minisat("1.cnf")
    #os.remove("1.cnf")
    if solution==None:
        print ("unsat!")
        exit(0)

    print_solution(solution)

main()
```

(https://smt.st/current_tree/puzzles/sudoku/SAT/sudoku_SAT.py)

The `make_distinct_bits_in_vector()` function preserves consistency of encoding.
The `make_distinct_vectors()` function makes 9 numbers unique.
The `cvt_vector_to_number()` decodes vector to number.
The `number_to_vector()` encodes number to vector.
The `main()` function has all necessary calls to make rows/columns/$3 \cdot 3$ boxes unique.

That works:

```
% python sudoku_SAT.py
len(clauses)= 12195
1 4 5 3 2 7 6 9 8
8 3 9 6 5 4 1 2 7
6 7 2 9 1 8 5 4 3
4 9 6 1 8 5 3 7 2
2 1 8 4 7 3 9 5 6
7 5 3 2 9 6 4 8 1
3 6 7 5 4 2 8 1 9
9 8 4 7 6 1 2 3 5
5 2 1 8 3 9 7 6 4
```

Same solution as earlier: 8.1.1.

Picosat tells this SAT instance has only one solution. Indeed, as they say, true Sudoku puzzle can have only one solution.

Measuring puzzle's hardness using MiniSat

Various puzzles printed in newspapers are divided by ”hardness” or number of clues. Let's see, how we can measure ”hardness”. SAT solver's clock time is not an options, because this is too easy problem for them. However, MiniSat can give other statistics:

35 clues, ”Easy” level from https://www.websudoku.com/?level=1.

"8.4.3.1.772.54.9....1.7.....39.....5.7.469.1.1.....79.....2.4....5.93.719.3.8.5.6"

```
restarts              : 1
```

```
conflicts              : 0              (0 /sec)
decisions              : 1              (0.00 % random) (99 /sec)
propagations           : 729            (72221 /sec)
conflict literals      : 0              (-nan % deleted)
```

29 clues, ”Medium” level from https://www.websudoku.com/?level=2.
"2......56.4..2.7.1.6.9..2......6..1...35784...9..3......4..7.6.3.9.5..2.17......4"

```
restarts               : 1
conflicts              : 0              (0 /sec)
decisions              : 1              (0.00 % random) (117 /sec)
propagations           : 729            (84945 /sec)
conflict literals      : 0              (-nan % deleted)
```

26 clues, ”Hard” level from https://www.websudoku.com/?level=3.
"..62..48.3..8....7.4..5........451.....9.2.....963........1..2.2....8..6.75..98.."

```
restarts               : 1
conflicts              : 0              (0 /sec)
decisions              : 1              (0.00 % random) (120 /sec)
propagations           : 729            (87431 /sec)
conflict literals      : 0              (-nan % deleted)
```

26 clues, ”Evil” level from https://www.websudoku.com/?level=4.
".......2..23..6..479.5.....3..6.45.....2.9.....73.5..6.....2.318..7..46..1......."

```
restarts               : 1
conflicts              : 1              (275 /sec)
decisions              : 2              (0.00 % random) (550 /sec)
propagations           : 810            (222711 /sec)
conflict literals      : 1              (0.00 % deleted)
```

23 clues, from http://www.norvig.com/sudoku.html, http://www.mirror.co.uk/news/weird-news/worlds-hardest-su
"..53.....8......2..7..1.5..4....53...1..7...6..32...8..6.5....9..4....3......97.."

```
restarts               : 1
conflicts              : 22             (2281 /sec)
decisions              : 30             (0.00 % random) (3110 /sec)
propagations           : 2516           (260861 /sec)
conflict literals      : 156            (21.61 % deleted)
```

These are ”conflicts”, ”decisions”, ”propagations”, ”conflict literals”.

Getting rid of one POPCNT1 function call

To make 9 unique 1..9 numbers we can use POPCNT1 function to make each row in matrix be unique and use *OR* boolean operation for all columns. That will have merely the same effect: all rows has to be unique to make each column to be evaluated to *True* if all variables in column are OR'ed. (I will do this in the next example: 8.2.4.)

That will make 3447 clauses instead of 12195, but somehow, SAT solvers works slower. No idea why.

8.2 Zebra puzzle (AKA Einstein puzzle)

8.2.1 SMT

Zebra puzzle is a popular puzzle, defined as follows:

1.There are five houses.
2.The Englishman lives in the red house.
3.The Spaniard owns the dog.
4.Coffee is drunk in the green house.
5.The Ukrainian drinks tea.
6.The green house is immediately to the right of the ivory house.

> 7.The Old Gold smoker owns snails.
> 8.Kools are smoked in the yellow house.
> 9.Milk is drunk in the middle house.
> 10.The Norwegian lives in the first house.
> 11.The man who smokes Chesterfields lives in the house next to the man with the fox.
> 12.Kools are smoked in the house next to the house where the horse is kept.
> 13.The Lucky Strike smoker drinks orange juice.
> 14.The Japanese smokes Parliaments.
> 15.The Norwegian lives next to the blue house.
>
> Now, who drinks water? Who owns the zebra?
>
> In the interest of clarity, it must be added that each of the five houses is painted a different color, and their inhabitants are of different national extractions, own different pets, drink different beverages and smoke different brands of American cigarets [sic]. One other thing: in statement 6, right means your right.

(https://en.wikipedia.org/wiki/Zebra_Puzzle)

It's a very good example of CSP[6].

We would encode each entity as integer variable, representing number of house.

Then, to define that Englishman lives in red house, we will add this constraint: `Englishman == Red`, meaning that number of a house where Englishmen resides and which is painted in red is the same.

To define that Norwegian lives next to the blue house, we don't really know, if it is at left side of blue house or at right side, but we know that house numbers are different by just 1. So we will define this constraint: `Norwegian==Blue-1 OR Norwegian==Blue+1`.

We will also need to limit all house numbers, so they will be in range of 1..5.

We will also use `Distinct` to show that all various entities of the same type are all has different house numbers.

```
#!/usr/bin/env python3

from z3 import *

Yellow, Blue, Red, Ivory, Green=Ints('Yellow Blue Red Ivory Green')
Norwegian, Ukrainian, Englishman, Spaniard, Japanese=Ints('Norwegian Ukrainian
    Englishman Spaniard Japanese')
Water, Tea, Milk, OrangeJuice, Coffee=Ints('Water Tea Milk OrangeJuice Coffee')
Kools, Chesterfield, OldGold, LuckyStrike, Parliament=Ints('Kools Chesterfield
    OldGold LuckyStrike Parliament')
Fox, Horse, Snails, Dog, Zebra=Ints('Fox Horse Snails Dog Zebra')

s = Solver()

# colors are distinct for all 5 houses:
s.add(Distinct(Yellow, Blue, Red, Ivory, Green))

# all nationalities are living in different houses:
s.add(Distinct(Norwegian, Ukrainian, Englishman, Spaniard, Japanese))

# so are beverages:
s.add(Distinct(Water, Tea, Milk, OrangeJuice, Coffee))

# so are cigarettes:
s.add(Distinct(Kools, Chesterfield, OldGold, LuckyStrike, Parliament))

# so are pets:
s.add(Distinct(Fox, Horse, Snails, Dog, Zebra))
```

[6] Constraint satisfaction problem

```
# limits.
# adding two constraints at once (separated by comma) is the same
# as adding one And() constraint with two subconstraints
s.add(Yellow>=1, Yellow<=5)
s.add(Blue>=1, Blue<=5)
s.add(Red>=1, Red<=5)
s.add(Ivory>=1, Ivory<=5)
s.add(Green>=1, Green<=5)

s.add(Norwegian>=1, Norwegian<=5)
s.add(Ukrainian>=1, Ukrainian<=5)
s.add(Englishman>=1, Englishman<=5)
s.add(Spaniard>=1, Spaniard<=5)
s.add(Japanese>=1, Japanese<=5)

s.add(Water>=1, Water<=5)
s.add(Tea>=1, Tea<=5)
s.add(Milk>=1, Milk<=5)
s.add(OrangeJuice>=1, OrangeJuice<=5)
s.add(Coffee>=1, Coffee<=5)

s.add(Kools>=1, Kools<=5)
s.add(Chesterfield>=1, Chesterfield<=5)
s.add(OldGold>=1, OldGold<=5)
s.add(LuckyStrike>=1, LuckyStrike<=5)
s.add(Parliament>=1, Parliament<=5)

s.add(Fox>=1, Fox<=5)
s.add(Horse>=1, Horse<=5)
s.add(Snails>=1, Snails<=5)
s.add(Dog>=1, Dog<=5)
s.add(Zebra>=1, Zebra<=5)

# main constraints of the puzzle:

# 2.The Englishman lives in the red house.
s.add(Englishman==Red)

# 3.The Spaniard owns the dog.
s.add(Spaniard==Dog)

# 4.Coffee is drunk in the green house.
s.add(Coffee==Green)

# 5.The Ukrainian drinks tea.
s.add(Ukrainian==Tea)

# 6.The green house is immediately to the right of the ivory house.
s.add(Green==Ivory+1)

# 7.The Old Gold smoker owns snails.
s.add(OldGold==Snails)

# 8.Kools are smoked in the yellow house.
s.add(Kools==Yellow)

# 9.Milk is drunk in the middle house.
s.add(Milk==3) # i.e., 3rd house

# 10.The Norwegian lives in the first house.
s.add(Norwegian==1)
```

```
# 11.The man who smokes Chesterfields lives in the house next to the man with the fox.
s.add(Or(Chesterfield==Fox+1, Chesterfield==Fox-1)) # left or right

# 12.Kools are smoked in the house next to the house where the horse is kept.
s.add(Or(Kools==Horse+1, Kools==Horse-1)) # left or right

# 13.The Lucky Strike smoker drinks orange juice.
s.add(LuckyStrike==OrangeJuice)

# 14.The Japanese smokes Parliaments.
s.add(Japanese==Parliament)

# 15.The Norwegian lives next to the blue house.
s.add(Or(Norwegian==Blue+1, Norwegian==Blue-1)) # left or right

r=s.check()
print (r)
if r==unsat:
    exit(0)
m=s.model()
print(m)
```

When we run it, we got correct result:

```
sat
[Snails = 3,
 Blue = 2,
 Ivory = 4,
 OrangeJuice = 4,
 Parliament = 5,
 Yellow = 1,
 Fox = 1,
 Zebra = 5,
 Horse = 2,
 Dog = 4,
 Tea = 2,
 Water = 1,
 Chesterfield = 2,
 Red = 3,
 Japanese = 5,
 LuckyStrike = 4,
 Norwegian = 1,
 Milk = 3,
 Kools = 1,
 OldGold = 3,
 Ukrainian = 2,
 Coffee = 5,
 Green = 5,
 Spaniard = 4,
 Englishman = 3]
```

8.2.2 KLEE

We just define all variables and add constraints:

```
#include "klee.h"

int main()
{
        int Yellow, Blue, Red, Ivory, Green;
```

```
    int Norwegian, Ukrainian, Englishman, Spaniard, Japanese;
    int Water, Tea, Milk, OrangeJuice, Coffee;
    int Kools, Chesterfield, OldGold, LuckyStrike, Parliament;
    int Fox, Horse, Snails, Dog, Zebra;

    klee_make_symbolic(&Yellow, sizeof(int), "Yellow");
    klee_make_symbolic(&Blue, sizeof(int), "Blue");
    klee_make_symbolic(&Red, sizeof(int), "Red");
    klee_make_symbolic(&Ivory, sizeof(int), "Ivory");
    klee_make_symbolic(&Green, sizeof(int), "Green");

    klee_make_symbolic(&Norwegian, sizeof(int), "Norwegian");
    klee_make_symbolic(&Ukrainian, sizeof(int), "Ukrainian");
    klee_make_symbolic(&Englishman, sizeof(int), "Englishman");
    klee_make_symbolic(&Spaniard, sizeof(int), "Spaniard");
    klee_make_symbolic(&Japanese, sizeof(int), "Japanese");

    klee_make_symbolic(&Water, sizeof(int), "Water");
    klee_make_symbolic(&Tea, sizeof(int), "Tea");
    klee_make_symbolic(&Milk, sizeof(int), "Milk");
    klee_make_symbolic(&OrangeJuice, sizeof(int), "OrangeJuice");
    klee_make_symbolic(&Coffee, sizeof(int), "Coffee");

    klee_make_symbolic(&Kools, sizeof(int), "Kools");
    klee_make_symbolic(&Chesterfield, sizeof(int), "Chesterfield");
    klee_make_symbolic(&OldGold, sizeof(int), "OldGold");
    klee_make_symbolic(&LuckyStrike, sizeof(int), "LuckyStrike");
    klee_make_symbolic(&Parliament, sizeof(int), "Parliament");

    klee_make_symbolic(&Fox, sizeof(int), "Fox");
    klee_make_symbolic(&Horse, sizeof(int), "Horse");
    klee_make_symbolic(&Snails, sizeof(int), "Snails");
    klee_make_symbolic(&Dog, sizeof(int), "Dog");
    klee_make_symbolic(&Zebra, sizeof(int), "Zebra");

    // limits.
    if (Yellow<1 || Yellow>5) return 0;
    if (Blue<1 || Blue>5) return 0;
    if (Red<1 || Red>5) return 0;
    if (Ivory<1 || Ivory>5) return 0;
    if (Green<1 || Green>5) return 0;

    if (Norwegian<1 || Norwegian>5) return 0;
    if (Ukrainian<1 || Ukrainian>5) return 0;
    if (Englishman<1 || Englishman>5) return 0;
    if (Spaniard<1 || Spaniard>5) return 0;
    if (Japanese<1 || Japanese>5) return 0;

    if (Water<1 || Water>5) return 0;
    if (Tea<1 || Tea>5) return 0;
    if (Milk<1 || Milk>5) return 0;
    if (OrangeJuice<1 || OrangeJuice>5) return 0;
    if (Coffee<1 || Coffee>5) return 0;

    if (Kools<1 || Kools>5) return 0;
    if (Chesterfield<1 || Chesterfield>5) return 0;
    if (OldGold<1 || OldGold>5) return 0;
    if (LuckyStrike<1 || LuckyStrike>5) return 0;
    if (Parliament<1 || Parliament>5) return 0;

    if (Fox<1 || Fox>5) return 0;
```

```
        if (Horse<1 || Horse>5) return 0;
        if (Snails<1 || Snails>5) return 0;
        if (Dog<1 || Dog>5) return 0;
        if (Zebra<1 || Zebra>5) return 0;

        // colors are distinct for all 5 houses:
        if (((1<<Yellow) | (1<<Blue) | (1<<Red) | (1<<Ivory) | (1<<Green))!=0x3E)
           return 0; // 111110

        // all nationalities are living in different houses:
        if (((1<<Norwegian) | (1<<Ukrainian) | (1<<Englishman) | (1<<Spaniard) | (1<<
           Japanese))!=0x3E) return 0; // 111110

        // so are beverages:
        if (((1<<Water) | (1<<Tea) | (1<<Milk) | (1<<OrangeJuice) | (1<<Coffee))!=0
           x3E) return 0; // 111110

        // so are cigarettes:
        if (((1<<Kools) | (1<<Chesterfield) | (1<<OldGold) | (1<<LuckyStrike) | (1<<
           Parliament))!=0x3E) return 0; // 111110

        // so are pets:
        if (((1<<Fox) | (1<<Horse) | (1<<Snails) | (1<<Dog) | (1<<Zebra))!=0x3E)
           return 0; // 111110

        // main constraints of the puzzle:

        // 2.The Englishman lives in the red house.
        if (Englishman!=Red) return 0;

        // 3.The Spaniard owns the dog.
        if (Spaniard!=Dog) return 0;

        // 4.Coffee is drunk in the green house.
        if (Coffee!=Green) return 0;

        // 5.The Ukrainian drinks tea.
        if (Ukrainian!=Tea) return 0;

        // 6.The green house is immediately to the right of the ivory house.
        if (Green!=Ivory+1) return 0;

        // 7.The Old Gold smoker owns snails.
        if (OldGold!=Snails) return 0;

        // 8.Kools are smoked in the yellow house.
        if (Kools!=Yellow) return 0;

        // 9.Milk is drunk in the middle house.
        if (Milk!=3) return 0; // i.e., 3rd house

        // 10.The Norwegian lives in the first house.
        if (Norwegian!=1) return 0;

        // 11.The man who smokes Chesterfields lives in the house next to the man
           with the fox.
        if (Chesterfield!=Fox+1 && Chesterfield!=Fox-1) return 0; // left or right

        // 12.Kools are smoked in the house next to the house where the horse is kept
           .
        if (Kools!=Horse+1 && Kools!=Horse-1) return 0; // left or right
        // 13.The Lucky Strike smoker drinks orange juice.
        if (LuckyStrike!=OrangeJuice) return 0;

        // 14.The Japanese smokes Parliaments.
        if (Japanese!=Parliament) return 0;

        // 15.The Norwegian lives next to the blue house.
        if (Norwegian!=Blue+1 && Norwegian!=Blue-1) return 0; // left or right

        // all constraints are satisfied at this point
        // force KLEE to produce .err file:
        klee_assert(0);

        return 0;
};
```

I force KLEE to find distinct values for colors, nationalities, cigarettes, etc, in the same way as I did for Sudoku earlier (8.1.1).

Let's run it:

```
% clang -emit-llvm -c -g -O0 -Xclang -disable-O0-optnone -I /tmp/klee_src/include/
   klee/ klee_zebra1.c
...

% time klee --libc=uclibc --optimize klee_zebra1.bc
KLEE: done: total instructions = 15399
KLEE: done: completed paths = 42
KLEE: done: partially completed paths = 1
KLEE: done: generated tests = 43
```

It works for ≈ 7 seconds on my Intel Core i3-3110M 2.4GHz notebook. Let's find out path, where `klee_assert()` has been executed:

```
% ls klee-last/*err
klee-last/test000037.external.err

% ktest-tool klee-last/test000037.ktest
ktest file : 'klee-last/test000037.ktest'
args       : ['klee_zebra1.bc']
num objects: 25
object  0: name: 'Yellow'
object  0: size: 4
object  0: data: b'\x01\x00\x00\x00'
object  0: hex : 0x01000000
object  0: int : 1
object  0: uint: 1
object  0: text: ....
object  1: name: 'Blue'
object  1: size: 4
object  1: data: b'\x02\x00\x00\x00'
object  1: hex : 0x02000000
object  1: int : 2
object  1: uint: 2
object  1: text: ....
object  2: name: 'Red'
object  2: size: 4
object  2: data: b'\x03\x00\x00\x00'
object  2: hex : 0x03000000
object  2: int : 3
object  2: uint: 3
object  2: text: ....
```

```
...

object 23: name: 'Dog'
object 23: size: 4
object 23: data: b'\x04\x00\x00\x00'
object 23: hex : 0x04000000
object 23: int : 4
object 23: uint: 4
object 23: text: ....
object 24: name: 'Zebra'
object 24: size: 4
object 24: data: b'\x05\x00\x00\x00'
object 24: hex : 0x05000000
object 24: int : 5
object 24: uint: 5
object 24: text: ....
```

This is indeed the correct solution.

How many tests are there?

```
% ls -la klee-last/*.ktest | wc -l
43
```

This is because KLEE found a test for each path, till each *return*.

How many *return* statement our program has, after all? Almost 43:

```
 % cat klee_zebra1.c| grep return | wc -l
45
```

---

klee_assume() also can be used:

```
#include "klee.h"

int main()
{
        int Yellow, Blue, Red, Ivory, Green;
        int Norwegian, Ukrainian, Englishman, Spaniard, Japanese;
        int Water, Tea, Milk, OrangeJuice, Coffee;
        int Kools, Chesterfield, OldGold, LuckyStrike, Parliament;
        int Fox, Horse, Snails, Dog, Zebra;

        klee_make_symbolic(&Yellow, sizeof(int), "Yellow");
        klee_make_symbolic(&Blue, sizeof(int), "Blue");
        klee_make_symbolic(&Red, sizeof(int), "Red");
        klee_make_symbolic(&Ivory, sizeof(int), "Ivory");
        klee_make_symbolic(&Green, sizeof(int), "Green");

        klee_make_symbolic(&Norwegian, sizeof(int), "Norwegian");
        klee_make_symbolic(&Ukrainian, sizeof(int), "Ukrainian");
        klee_make_symbolic(&Englishman, sizeof(int), "Englishman");
        klee_make_symbolic(&Spaniard, sizeof(int), "Spaniard");
        klee_make_symbolic(&Japanese, sizeof(int), "Japanese");

        klee_make_symbolic(&Water, sizeof(int), "Water");
        klee_make_symbolic(&Tea, sizeof(int), "Tea");
        klee_make_symbolic(&Milk, sizeof(int), "Milk");
        klee_make_symbolic(&OrangeJuice, sizeof(int), "OrangeJuice");
        klee_make_symbolic(&Coffee, sizeof(int), "Coffee");

        klee_make_symbolic(&Kools, sizeof(int), "Kools");
        klee_make_symbolic(&Chesterfield, sizeof(int), "Chesterfield");
        klee_make_symbolic(&OldGold, sizeof(int), "OldGold");
        klee_make_symbolic(&LuckyStrike, sizeof(int), "LuckyStrike");
        klee_make_symbolic(&Parliament, sizeof(int), "Parliament");

        klee_make_symbolic(&Fox, sizeof(int), "Fox");
        klee_make_symbolic(&Horse, sizeof(int), "Horse");
        klee_make_symbolic(&Snails, sizeof(int), "Snails");
        klee_make_symbolic(&Dog, sizeof(int), "Dog");
        klee_make_symbolic(&Zebra, sizeof(int), "Zebra");

        // limits.
        klee_assume (Yellow>=1 && Yellow<=5);
        klee_assume (Blue>=1 && Blue<=5);
        klee_assume (Red>=1 && Red<=5);
        klee_assume (Ivory>=1 && Ivory<=5);
        klee_assume (Green>=1 && Green<=5);

        klee_assume (Norwegian>=1 && Norwegian<=5);
        klee_assume (Ukrainian>=1 && Ukrainian<=5);
        klee_assume (Englishman>=1 && Englishman<=5);
        klee_assume (Spaniard>=1 && Spaniard<=5);
        klee_assume (Japanese>=1 && Japanese<=5);

        klee_assume (Water>=1 && Water<=5);
        klee_assume (Tea>=1 && Tea<=5);
        klee_assume (Milk>=1 && Milk<=5);
        klee_assume (OrangeJuice>=1 && OrangeJuice<=5);
        klee_assume (Coffee>=1 && Coffee<=5);

        klee_assume (Kools>=1 && Kools<=5);
        klee_assume (Chesterfield>=1 && Chesterfield<=5);
        klee_assume (OldGold>=1 && OldGold<=5);
        klee_assume (LuckyStrike>=1 && LuckyStrike<=5);
        klee_assume (Parliament>=1 && Parliament<=5);

        klee_assume (Fox>=1 && Fox<=5);
        klee_assume (Horse>=1 && Horse<=5);
        klee_assume (Snails>=1 && Snails<=5);
        klee_assume (Dog>=1 && Dog<=5);
        klee_assume (Zebra>=1 && Zebra<=5);

        // colors are distinct for all 5 houses:
        klee_assume (((1<<Yellow) | (1<<Blue) | (1<<Red) | (1<<Ivory) | (1<<Green))
           ==0x3E); // 111110

        // all nationalities are living in different houses:
        klee_assume (((1<<Norwegian) | (1<<Ukrainian) | (1<<Englishman) | (1<<
           Spaniard) | (1<<Japanese))==0x3E); // 111110

        // so are beverages:
        klee_assume (((1<<Water) | (1<<Tea) | (1<<Milk) | (1<<OrangeJuice) | (1<<
           Coffee))==0x3E); // 111110

        // so are cigarettes:
        klee_assume (((1<<Kools) | (1<<Chesterfield) | (1<<OldGold) | (1<<LuckyStrike
           ) | (1<<Parliament))==0x3E); // 111110

        // so are pets:
        klee_assume (((1<<Fox) | (1<<Horse) | (1<<Snails) | (1<<Dog) | (1<<Zebra))==0
           x3E); // 111110
        // main constraints of the puzzle:

        // 2.The Englishman lives in the red house.
        klee_assume (Englishman==Red);

        // 3.The Spaniard owns the dog.
        klee_assume (Spaniard==Dog);

        // 4.Coffee is drunk in the green house.
        klee_assume (Coffee==Green);

        // 5.The Ukrainian drinks tea.
        klee_assume (Ukrainian==Tea);

        // 6.The green house is immediately to the right of the ivory house.
        klee_assume (Green==Ivory+1);

        // 7.The Old Gold smoker owns snails.
        klee_assume (OldGold==Snails);

        // 8.Kools are smoked in the yellow house.
        klee_assume (Kools==Yellow);

        // 9.Milk is drunk in the middle house.
        klee_assume (Milk==3); // i.e., 3rd house

        // 10.The Norwegian lives in the first house.
        klee_assume (Norwegian==1);

        // 11.The man who smokes Chesterfields lives in the house next to the man
           with the fox.
        klee_assume (Chesterfield==Fox+1 || Chesterfield==Fox-1); // left or right

        // 12.Kools are smoked in the house next to the house where the horse is kept
           .
        klee_assume (Kools==Horse+1 || Kools==Horse-1); // left or right

        // 13.The Lucky Strike smoker drinks orange juice.
        klee_assume (LuckyStrike==OrangeJuice);

        // 14.The Japanese smokes Parliaments.
        klee_assume (Japanese==Parliament);

        // 15.The Norwegian lives next to the blue house.
        klee_assume (Norwegian==Blue+1 || Norwegian==Blue-1); // left or right

        // all constraints are satisfied at this point
        // force KLEE to produce .err file:
        klee_assert(0);
};
```

```
% time klee --libc=uclibc --optimize klee_zebra2.bc
...
KLEE: done: total instructions = 12304
KLEE: done: completed paths = 0
KLEE: done: partially completed paths = 1
KLEE: done: generated tests = 1

% ls klee-last/
assembly.ll  messages.txt  run.stats           test000001.external.err  test000001.
   ktest
```

```
info          run.istats    solver-queries.smt2  test000001.kquery        warnings.txt
```

...and this version works slightly faster (≈ 5 seconds). Faster, because KLEE has to find only one path, till the end of the function. Hence, here is only one test generated.

8.2.3 CBMC

CBMC solution is simpler, we don't define, which variables are subject to search.

```
#include <assert.h>

int check()
{
        int Yellow, Blue, Red, Ivory, Green;
        int Norwegian, Ukrainian, Englishman, Spaniard, Japanese;
        int Water, Tea, Milk, OrangeJuice, Coffee;
        int Kools, Chesterfield, OldGold, LuckyStrike, Parliament;
        int Fox, Horse, Snails, Dog, Zebra;

        // limits.
        if (Yellow<1 || Yellow>5) return 0;
        if (Blue<1 || Blue>5) return 0;
        if (Red<1 || Red>5) return 0;
        if (Ivory<1 || Ivory>5) return 0;
        if (Green<1 || Green>5) return 0;

        if (Norwegian<1 || Norwegian>5) return 0;
        if (Ukrainian<1 || Ukrainian>5) return 0;
        if (Englishman<1 || Englishman>5) return 0;
        if (Spaniard<1 || Spaniard>5) return 0;
        if (Japanese<1 || Japanese>5) return 0;

        if (Water<1 || Water>5) return 0;
        if (Tea<1 || Tea>5) return 0;
        if (Milk<1 || Milk>5) return 0;
        if (OrangeJuice<1 || OrangeJuice>5) return 0;
        if (Coffee<1 || Coffee>5) return 0;

        if (Kools<1 || Kools>5) return 0;
        if (Chesterfield<1 || Chesterfield>5) return 0;
        if (OldGold<1 || OldGold>5) return 0;
        if (LuckyStrike<1 || LuckyStrike>5) return 0;
        if (Parliament<1 || Parliament>5) return 0;

        if (Fox<1 || Fox>5) return 0;
        if (Horse<1 || Horse>5) return 0;
        if (Snails<1 || Snails>5) return 0;
        if (Dog<1 || Dog>5) return 0;
        if (Zebra<1 || Zebra>5) return 0;

        // colors are distinct for all 5 houses:
        if (((1<<Yellow) | (1<<Blue) | (1<<Red) | (1<<Ivory) | (1<<Green))!=0x3E)
           return 0; // 111110

        // all nationalities are living in different houses:
        if (((1<<Norwegian) | (1<<Ukrainian) | (1<<Englishman) | (1<<Spaniard) | (1<<
           Japanese))!=0x3E) return 0; // 111110

        // so are beverages:
        if (((1<<Water) | (1<<Tea) | (1<<Milk) | (1<<OrangeJuice) | (1<<Coffee))!=0
           x3E) return 0; // 111110
        // so are cigarettes:
        if (((1<<Kools) | (1<<Chesterfield) | (1<<OldGold) | (1<<LuckyStrike) | (1<<
           Parliament))!=0x3E) return 0; // 111110

        // so are pets:
        if (((1<<Fox) | (1<<Horse) | (1<<Snails) | (1<<Dog) | (1<<Zebra))!=0x3E)
           return 0; // 111110

        // main constraints of the puzzle:

        // 2.The Englishman lives in the red house.
        if (Englishman!=Red) return 0;

        // 3.The Spaniard owns the dog.
        if (Spaniard!=Dog) return 0;

        // 4.Coffee is drunk in the green house.
        if (Coffee!=Green) return 0;

        // 5.The Ukrainian drinks tea.
        if (Ukrainian!=Tea) return 0;

        // 6.The green house is immediately to the right of the ivory house.
        if (Green!=Ivory+1) return 0;

        // 7.The Old Gold smoker owns snails.
        if (OldGold!=Snails) return 0;

        // 8.Kools are smoked in the yellow house.
        if (Kools!=Yellow) return 0;

        // 9.Milk is drunk in the middle house.
        if (Milk!=3) return 0; // i.e., 3rd house

        // 10.The Norwegian lives in the first house.
        if (Norwegian!=1) return 0;

        // 11.The man who smokes Chesterfields lives in the house next to the man
           with the fox.
        if (Chesterfield!=Fox+1 && Chesterfield!=Fox-1) return 0; // left or right

        // 12.Kools are smoked in the house next to the house where the horse is kept
           .
        if (Kools!=Horse+1 && Kools!=Horse-1) return 0; // left or right

        // 13.The Lucky Strike smoker drinks orange juice.
        if (LuckyStrike!=OrangeJuice) return 0;

        // 14.The Japanese smokes Parliaments.
        if (Japanese!=Parliament) return 0;

        // 15.The Norwegian lives next to the blue house.
        if (Norwegian!=Blue+1 && Norwegian!=Blue-1) return 0; // left or right

        // all constraints are satisfied at this point
        assert(0);

        return 0;
};
```

```
% cbmc --trace --function check CBMC_zebra.c
```

```
...

** Results:
CBMC_zebra.c function check
[check.assertion.1] line 102 assertion 0: FAILURE

Trace for check.assertion.1:

State 21 file CBMC_zebra.c function check line 5 thread 0
-----------------------------------------------------
  Yellow=1 (00000000 00000000 00000000 00000001)

State 22 file CBMC_zebra.c function check line 5 thread 0
-----------------------------------------------------
  Blue=2 (00000000 00000000 00000000 00000010)

State 23 file CBMC_zebra.c function check line 5 thread 0
-----------------------------------------------------
  Red=3 (00000000 00000000 00000000 00000011)

State 24 file CBMC_zebra.c function check line 5 thread 0
-----------------------------------------------------
  Ivory=4 (00000000 00000000 00000000 00000100)

State 25 file CBMC_zebra.c function check line 5 thread 0
-----------------------------------------------------
  Green=5 (00000000 00000000 00000000 00000101)

State 26 file CBMC_zebra.c function check line 6 thread 0
-----------------------------------------------------
  Norwegian=1 (00000000 00000000 00000000 00000001)

State 27 file CBMC_zebra.c function check line 6 thread 0
-----------------------------------------------------
  Ukrainian=2 (00000000 00000000 00000000 00000010)

State 28 file CBMC_zebra.c function check line 6 thread 0
-----------------------------------------------------
  Englishman=3 (00000000 00000000 00000000 00000011)

State 29 file CBMC_zebra.c function check line 6 thread 0
-----------------------------------------------------
  Spaniard=4 (00000000 00000000 00000000 00000100)

State 30 file CBMC_zebra.c function check line 6 thread 0
-----------------------------------------------------
  Japanese=5 (00000000 00000000 00000000 00000101)

State 31 file CBMC_zebra.c function check line 7 thread 0
-----------------------------------------------------
  Water=1 (00000000 00000000 00000000 00000001)

State 32 file CBMC_zebra.c function check line 7 thread 0
-----------------------------------------------------
  Tea=2 (00000000 00000000 00000000 00000010)

State 33 file CBMC_zebra.c function check line 7 thread 0
-----------------------------------------------------
  Milk=3 (00000000 00000000 00000000 00000011)
```

```
State 34 file CBMC_zebra.c function check line 7 thread 0
----------------------------------------------------
  OrangeJuice=4 (00000000 00000000 00000000 00000100)

State 35 file CBMC_zebra.c function check line 7 thread 0
----------------------------------------------------
  Coffee=5 (00000000 00000000 00000000 00000101)

State 36 file CBMC_zebra.c function check line 8 thread 0
----------------------------------------------------
  Kools=1 (00000000 00000000 00000000 00000001)

State 37 file CBMC_zebra.c function check line 8 thread 0
----------------------------------------------------
  Chesterfield=2 (00000000 00000000 00000000 00000010)

State 38 file CBMC_zebra.c function check line 8 thread 0
----------------------------------------------------
  OldGold=3 (00000000 00000000 00000000 00000011)

State 39 file CBMC_zebra.c function check line 8 thread 0
----------------------------------------------------
  LuckyStrike=4 (00000000 00000000 00000000 00000100)

State 40 file CBMC_zebra.c function check line 8 thread 0
----------------------------------------------------
  Parliament=5 (00000000 00000000 00000000 00000101)

State 41 file CBMC_zebra.c function check line 9 thread 0
----------------------------------------------------
  Fox=1 (00000000 00000000 00000000 00000001)

State 42 file CBMC_zebra.c function check line 9 thread 0
----------------------------------------------------
  Horse=2 (00000000 00000000 00000000 00000010)

State 43 file CBMC_zebra.c function check line 9 thread 0
----------------------------------------------------
  Snails=3 (00000000 00000000 00000000 00000011)

State 44 file CBMC_zebra.c function check line 9 thread 0
----------------------------------------------------
  Dog=4 (00000000 00000000 00000000 00000100)

State 45 file CBMC_zebra.c function check line 9 thread 0
----------------------------------------------------
  Zebra=5 (00000000 00000000 00000000 00000101)

Violated property:
  file CBMC_zebra.c function check line 102 thread 0
  assertion 0
  0 != 0



** 1 of 1 failed (2 iterations)
VERIFICATION FAILED
```

The solution is no different, of course.

8.2.4 Zebra puzzle as a SAT problem

I would define each variable as vector of 5 variables, as I did before in Sudoku solver: 8.1.6.

I also use POPCNT1 function, but unlike previous example, I used Wolfram Mathematica to generate it in CNF form:

```
In[]:= tbl1=Table[PadLeft[IntegerDigits[i,2],5] ->If[Equal[DigitCount[i
    ,2][[1]],1],1,0],{i,0,63}]
Out[]= {{0,0,0,0,0}->0,
{0,0,0,0,1}->1,
{0,0,0,1,0}->1,
{0,0,0,1,1}->0,
{0,0,1,0,0}->1,
{0,0,1,0,1}->0,

...

{1,1,1,1,0}->0,
{1,1,1,1,1}->0}

In[]:= BooleanConvert[BooleanFunction[tbl1,{a,b,c,d,e}],"CNF"]
Out[]= (!a||!b)&&(!a||!c)&&(!a||!d)&&(!a||!e)&&(a||b||c||d||e)&&(!b||!c)&&(!b||!d)
    &&(!b||!e)&&(!c||!d)&&(!c||!e)&&(!d||!e)
```

Also, as I suggested before (8.1.6), I used *OR* operation as the second step.

```
def mathematica_to_CNF (s, d):
    for k in d.keys():
        s=s.replace(k, d[k])
    s=s.replace("!", "-").replace("||", " ").replace("(", "").replace(")", "")
    s=s.split ("&&")
    return s

def add_popcnt1(v1, v2, v3, v4, v5):
    global clauses
    s="(!a||!b)&&" \
      "(!a||!c)&&" \
      "(!a||!d)&&" \
      "(!a||!e)&&" \
      "(!b||!c)&&" \
      "(!b||!d)&&" \
      "(!b||!e)&&" \
      "(!c||!d)&&" \
      "(!c||!e)&&" \
      "(!d||!e)&&" \
      "(a||b||c||d||e)"

    clauses=clauses+mathematica_to_CNF(s, {"a":v1, "b":v2, "c":v3, "d":v4, "e":v5})

...

# k=tuple: ("high-level" variable name, number of bit (0..4))
# v=variable number in CNF
vars={}
vars_last=1

...

def alloc_distinct_variables(names):
    global vars
    global vars_last
    for name in names:
        for i in range(5):
            vars[(name,i)]=str(vars_last)
            vars_last=vars_last+1

        add_popcnt1(vars[(name,0)], vars[(name,1)], vars[(name,2)], vars[(name,3)],
           vars[(name,4)])

    # make them distinct:
    for i in range(5):
        clauses.append(vars[(names[0],i)] + " " + vars[(names[1],i)] + " " + vars[(
           names[2],i)] + " " + vars[(names[3],i)] + " " + vars[(names[4],i)])

...

alloc_distinct_variables(["Yellow", "Blue", "Red", "Ivory", "Green"])
alloc_distinct_variables(["Norwegian", "Ukrainian", "Englishman", "Spaniard", "
   Japanese"])
alloc_distinct_variables(["Water", "Tea", "Milk", "OrangeJuice", "Coffee"])
alloc_distinct_variables(["Kools", "Chesterfield", "OldGold", "LuckyStrike", "
   Parliament"])
alloc_distinct_variables(["Fox", "Horse", "Snails", "Dog", "Zebra"])

...
```

Now we have 5 boolean variables for each *high-level* variable, and each group of variables will always have distinct values.

Now let's reread puzzle description: "2.The Englishman lives in the red house.". That's easy. In my Z3 and KLEE examples I just wrote "Englishman==Red". Same story here: we just add a clauses showing that 5 boolean variables for "Englishman" must be equal to 5 booleans for "Red".

On a lowest CNF level, if we want to say that two variables must be equal to each other, we add two clauses:
$(var1 \lor \neg var2) \land (\neg var1 \lor var2)$

That means, both *var1* and *var2* values must be *False* or *True*, but they cannot be different.

```
def add_eq_clauses(var1, var2):
    global clauses
    clauses.append(var1 + " -" + var2)
    clauses.append("-"+var1 + " " + var2)

def add_eq (n1, n2):
    for i in range(5):
        add_eq_clauses(vars[(n1,i)], vars[(n2, i)])

...

# 2.The Englishman lives in the red house.
add_eq("Englishman","Red")

# 3.The Spaniard owns the dog.
add_eq("Spaniard","Dog")

# 4.Coffee is drunk in the green house.
add_eq("Coffee","Green")

...
```

Now the next conditions: "9.Milk is drunk in the middle house." (i.e., 3rd house), "10.The Norwegian lives in the first house." We can just assign boolean values directly:

```
# n=1..5
def add_eq_var_n (name, n):
    global clauses
    global vars
    for i in range(5):
        if i==n-1:
            clauses.append(vars[(name,i)]) # always True
        else:
            clauses.append("-"+vars[(name,i)]) # always False

...

# 9.Milk is drunk in the middle house.
add_eq_var_n("Milk",3) # i.e., 3rd house

# 10.The Norwegian lives in the first house.
add_eq_var_n("Norwegian",1)
```

For “Milk” we will have “0 0 1 0 0” value, for “Norwegian”: “1 0 0 0 0”.

What to do with this? “6.The green house is immediately to the right of the ivory house.” I can construct the following condition:

```
    Ivory      Green
AND(1 0 0 0 0  0 1 0 0 0)
.. OR ..
AND(0 1 0 0 0  0 0 1 0 0)
.. OR ..
AND(0 0 1 0 0  0 0 0 1 0)
.. OR ..
AND(0 0 0 1 0  0 0 0 0 1)
```

There is no “0 0 0 0 1” for “Ivory”, because it cannot be the last one. Now I can convert these conditions to CNF using Wolfram Mathematica:

```
In[]:= BooleanConvert[(a1&& !b1&&!c1&&!d1&&!e1&&!a2&& b2&&!c2&&!d2&&!e2) ||
(!a1&& b1&&!c1&&!d1&&!e1&&!a2&& !b2&&c2&&!d2&&!e2) ||
(!a1&& !b1&&c1&&!d1&&!e1&&!a2&& !b2&&!c2&&d2&&!e2) ||
(!a1&& !b1&&!c1&&d1&&!e1&&!a2&& !b2&&!c2&&!d2&&e2) ,"CNF"]

Out[]= (!a1||!b1)&&(!a1||!c1)&&(!a1||!d1)&&(a1||b1||c1||d1)&&!a2&&(!b1||!b2)&&(!b1||!
    c1)&&
(!b1||!d1)&&(b1||b2||c1||d1)&&(!b2||!c1)&&(!b2||!c2)&&(!b2||!d1)&&(!b2||!d2)&&(!b2||!
    e2)&&
(b2||c1||c2||d1)&&(b2||c2||d1||d2)&&(b2||c2||d2||e2)&&(!c1||!c2)&&(!c1||!d1)&&(!c2||!
    d1)&&
(!c2||!d2)&&(!c2||!e2)&&(!d1||!d2)&&(!d2||!e2)&&!e1
```

And here is a piece of my Python code:

```
def add_right (n1, n2):
    global clauses
    s="(!a1||!b1)&&(!a1||!c1)&&(!a1||!d1)&&(a1||b1||c1||d1)&&!a2&&(!b1||!b2)&&(!b1||!
        c1)&&(!b1||!d1)&&" \
      "(b1||b2||c1||d1)&&(!b2||!c1)&&(!b2||!c2)&&(!b2||!d1)&&(!b2||!d2)&&(!b2||!e2)
         &&(b2||c1||c2||d1)&&" \
      "(b2||c2||d1||d2)&&(b2||c2||d2||e2)&&(!c1||!c2)&&(!c1||!d1)&&(!c2||!d1)&&(!c2
         ||!d2)&&(!c2||!e2)&&" \
      "(!d1||!d2)&&(!d2||!e2)&&!e1"

    clauses=clauses+mathematica_to_CNF(s, {
        "a1": vars[(n1,0)], "b1": vars[(n1,1)], "c1": vars[(n1,2)], "d1": vars[(n1,3)
           ], "e1": vars[(n1,4)],
        "a2": vars[(n2,0)], "b2": vars[(n2,1)], "c2": vars[(n2,2)], "d2": vars[(n2,3)
           ], "e2": vars[(n2,4)]})

...

# 6.The green house is immediately to the right of the ivory house.
```

```
add_right("Ivory", "Green")
```

What we will do with that? “11.The man who smokes Chesterfields lives in the house next to the man with the fox.” “12.Kools are smoked in the house next to the house where the horse is kept.”

We don’t know side, left or right, but we know that they are differ in one. Here is a clauses I would add:

```
    Chesterfield  Fox
AND(0 0 0 0 1     0 0 0 1 0)
.. OR ..
AND(0 0 0 1 0     0 0 0 0 1)
AND(0 0 0 1 0     0 0 1 0 0)
.. OR ..
AND(0 0 1 0 0     0 1 0 0 0)
AND(0 0 1 0 0     0 0 0 1 0)
.. OR ..
AND(0 1 0 0 0     1 0 0 0 0)
AND(0 1 0 0 0     0 0 1 0 0)
.. OR ..
AND(1 0 0 0 0     0 1 0 0 0)
```

I can convert this into CNF using Mathematica again:

```
In[]:= BooleanConvert[(a1&& !b1&&!c1&&!d1&&!e1&&!a2&& b2&&!c2&&!d2&&!e2) ||

(!a1&& b1&&!c1&&!d1&&!e1&&a2&& !b2&&!c2&&!d2&&!e2) ||
(!a1&& b1&&!c1&&!d1&&!e1&&!a2&& !b2&&c2&&!d2&&!e2) ||

(!a1&& !b1&&c1&&!d1&&!e1&&!a2&& b2&&!c2&&!d2&&!e2) ||
(!a1&& !b1&&c1&&!d1&&!e1&&!a2&& !b2&&!c2&&d2&&!e2) ||

(!a1&& !b1&&!c1&&d1&&!e1&&!a2&& !b2&&c2&&!d2&&!e2) ||
(!a1&& !b1&&!c1&&d1&&!e1&&!a2&& !b2&&!c2&&!d2&&e2) ||

(!a1&& !b1&&!c1&&!d1&&e1&&!a2&& !b2&&!c2&&d2&&!e2) ,"CNF"]

Out[]= (!a1||!b1)&&(!a1||!c1)&&(!a1||!d1)&&(!a1||!e1)&&(a1||b1||c1||d1||e1)&&(!a2||b1
    )&&(!a2||!b2)&&
(!a2||!c2)&&(!a2||!d2)&&(!a2||!e2)&&(a2||b2||c1||c2||d1||e1)&&(a2||b2||c2||d1||d2)&&(
    a2||b2||c2||d2||e2)&&
(!b1||!b2)&&(!b1||!c1)&&(!b1||!d1)&&(!b1||!e1)&&(b1||b2||c1||d1||e1)&&(!b2||!c2)&&(!
    b2||!d1)&&(!b2||!d2)&&
(!b2||!e1)&&(!b2||!e2)&&(!c1||!c2)&&(!c1||!d1)&&(!c1||!e1)&&(!c2||!d2)&&(!c2||!e1)
    &&(!c2||!e2)&&
(!d1||!d2)&&(!d1||!e1)&&(!d2||!e2)
```

And here is my code:

```
def add_right_or_left (n1, n2):
    global clauses
    s="(!a1||!b1)&&(!a1||!c1)&&(!a1||!d1)&&(!a1||!e1)&&(a1||b1||c1||d1||e1)&&(!a2||b1
        )&&" \
      "(!a2||!b2)&&(!a2||!c2)&&(!a2||!d2)&&(!a2||!e2)&&(a2||b2||c1||c2||d1||e1)&&(a2
          ||b2||c2||d1||d2)&&" \
       "(a2||b2||c2||d2||e2)&&(!b1||!b2)&&(!b1||!c1)&&(!b1||!d1)&&(!b1||!e1)&&(b1||b2
          ||c1||d1||e1)&&" \
       "(!b2||!c2)&&(!b2||!d1)&&(!b2||!d2)&&(!b2||!e1)&&(!b2||!e2)&&(!c1||!c2)&&(!c1
          ||!d1)&&(!c1||!e1)&&" \
       "(!c2||!d2)&&(!c2||!e1)&&(!c2||!e2)&&(!d1||!d2)&&(!d1||!e1)&&(!d2||!e2)"

    clauses=clauses+mathematica_to_CNF(s, {
        "a1": vars[(n1,0)], "b1": vars[(n1,1)], "c1": vars[(n1,2)], "d1": vars[(n1,3)
          ], "e1": vars[(n1,4)],
        "a2": vars[(n2,0)], "b2": vars[(n2,1)], "c2": vars[(n2,2)], "d2": vars[(n2,3)
          ], "e2": vars[(n2,4)]})
```

```
...

# 11.The man who smokes Chesterfields lives in the house next to the man with the fox.
add_right_or_left("Chesterfield","Fox") # left or right

# 12.Kools are smoked in the house next to the house where the horse is kept.
add_right_or_left("Kools","Horse") # left or right
```

This is it! The full source code: https://smt.st/current_tree/puzzles/zebra/SAT/zebra_SAT.py.

Resulting CNF instance has 125 boolean variables and 511 clauses: https://smt.st/current_tree/puzzles/zebra/SAT/1.cnf. It is a piece of cake for any SAT solver. Even my toy-level SAT-solver (23.1) can solve it in ~1 second on my ancient Intel Atom netbook.

And of course, there is only one possible solution, what is acknowledged by Picosat.

```
% python zebra_SAT.py
Yellow 1
Blue 2
Red 3
Ivory 4
Green 5
Norwegian 1
Ukrainian 2
Englishman 3
Spaniard 4
Japanese 5
Water 1
Tea 2
Milk 3
OrangeJuice 4
Coffee 5
Kools 1
Chesterfield 2
OldGold 3
LuckyStrike 4
Parliament 5
Fox 1
Horse 2
Snails 3
Dog 4
Zebra 5
```

8.3 Solving pipe puzzle using Z3 SMT-solver

"Pipe puzzle" is a popular puzzle (just google "pipe puzzle" and look at images).

This is how shuffled puzzle looks like:

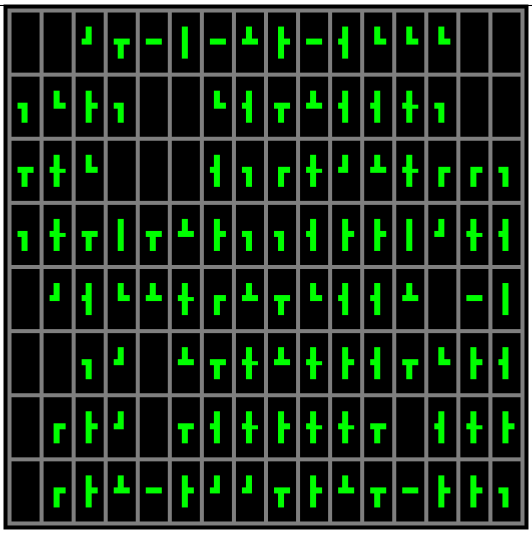


Figure 8.3: Shuffled puzzle

...and solved:

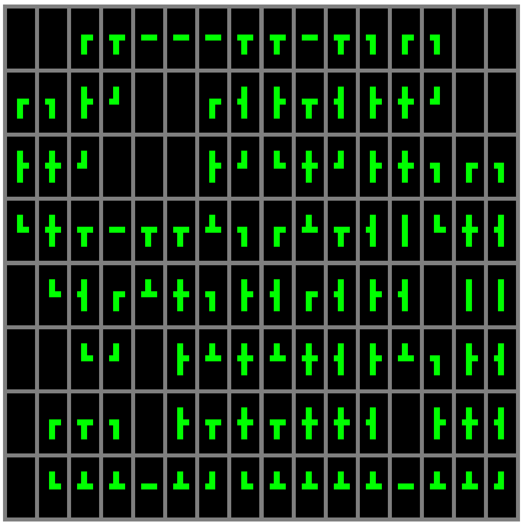
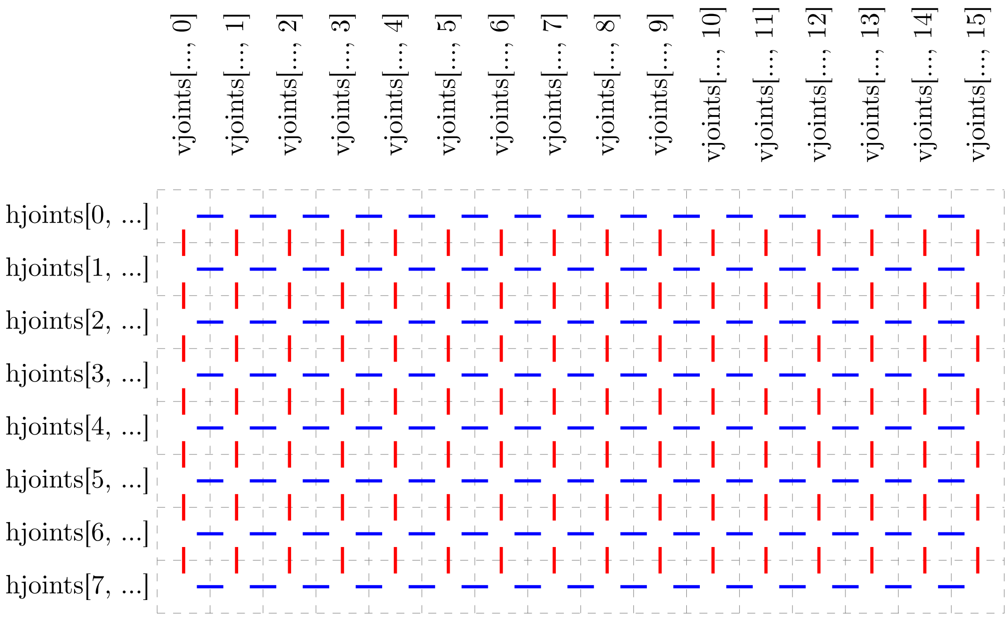


Figure 8.4: Solved puzzle

Let's try to find a way to solve it.

8.3.1 Generation

First, we need to generate it. Here is my quick idea on it. Take 8*16 array of cells. Each cell may contain some type of block. There are joints between cells:



Blue lines are horizontal joints, red lines are vertical joints. We just set each joint to 0/false (absent) or 1/true (present), randomly.

Once set, it's now easy to find type for each cell. There are:

joints	our internal name	angle	symbol
0	type 0	0°	(space)
2	type 2a	0°	┃
2	type 2a	90°	━
2	type 2b	0°	┏
2	type 2b	90°	┛
2	type 2b	180°	┛
2	type 2b	270°	┗
3	type 3	0°	┣
3	type 3	90°	┳
3	type 3	180°	┫
3	type 3	270°	┻
4	type 4	0°	╋

Dangling joints can be preset at a first stage (i.e., cell with only one joint), but they are removed recursively, these cells are transforming into empty cells. Hence, at the end, all cells has at least two joints, and the whole plumbing system has no connections with outer world—I hope this would make things clearer.

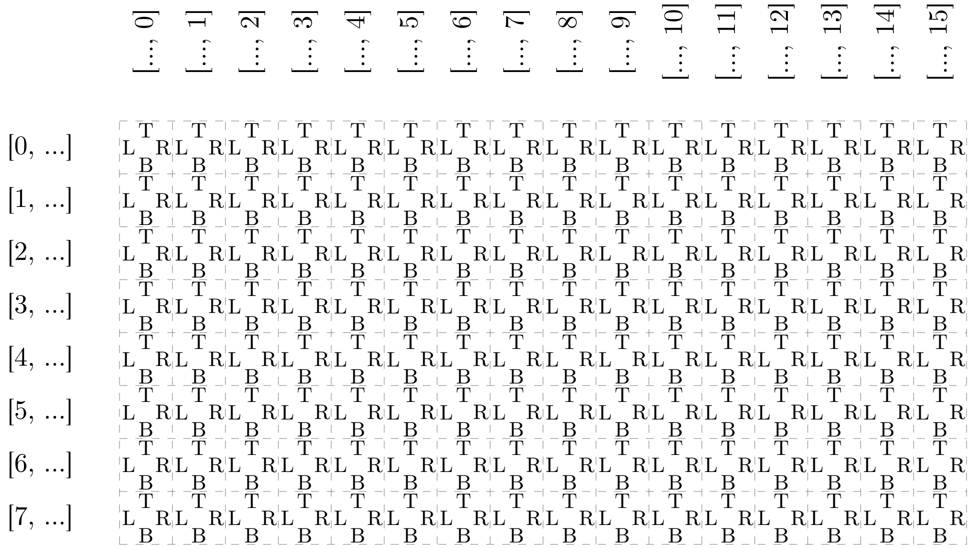
The C source code of generator is here: https://smt.st/current_tree/puzzles/pipe/generator. All horizontal joints are stored in the global array *hjoints[]* and vertical in *vjoints[]*.

The C program generates ANSI-colored output like it has been showed above (8.3, 8.4) plus an array of types, with no angle information about each cell:

```
[
["0", "0", "2b", "3", "2a", "2a", "2a", "3", "3", "2a", "3", "2b", "2b", "2b", "0", "0"],
["2b", "2b", "3", "2b", "0", "0", "2b", "3", "3", "3", "3", "3", "4", "2b", "0", "0"],
["3", "4", "2b", "0", "0", "0", "3", "2b", "2b", "4", "2b", "3", "4", "2b", "2b", "2b"],
["2b", "4", "3", "2a", "3", "3", "3", "2b", "2b", "3", "3", "3", "2a", "2b", "4", "3"],
["0", "2b", "3", "2b", "3", "4", "2b", "3", "3", "2b", "3", "3", "3", "0", "2a", "2a"],
["0", "0", "2b", "2b", "0", "3", "3", "4", "3", "4", "3", "3", "3", "2b", "3", "3"],
["0", "2b", "3", "2b", "0", "3", "3", "4", "3", "4", "4", "3", "0", "3", "4", "3"],
["0", "2b", "3", "3", "2a", "3", "2b", "2b", "3", "3", "3", "3", "2a", "3", "3", "2b"],
]
```

8.3.2 Solving

First of all, we would think about 8*16 array of cells, where each has four bits: “T” (top), “B” (bottom), “L” (left), “R” (right). Each bit represents half of joint.



Now we define arrays of each of four half-joints + angle information:

```
HEIGHT=8
WIDTH=16

# if T/B/R/L is Bool instead of Int, Z3 solver will work faster
T=[[Bool('cell_%d_%d_top' % (r, c)) for c in range(WIDTH)] for r in range(HEIGHT)]
B=[[Bool('cell_%d_%d_bottom' % (r, c)) for c in range(WIDTH)] for r in range(HEIGHT)]
R=[[Bool('cell_%d_%d_right' % (r, c)) for c in range(WIDTH)] for r in range(HEIGHT)]
L=[[Bool('cell_%d_%d_left' % (r, c)) for c in range(WIDTH)] for r in range(HEIGHT)]
A=[[Int('cell_%d_%d_angle' % (r, c)) for c in range(WIDTH)] for r in range(HEIGHT)]
```

We know that if each of half-joints is present, corresponding half-joint must be also present, and vice versa. We define this using these constraints:

```
# shorthand variables for True and False:
t=True
f=False

# "top" of each cell must be equal to "bottom" of the cell above
# "bottom" of each cell must be equal to "top" of the cell below
# "left" of each cell must be equal to "right" of the cell at left
# "right" of each cell must be equal to "left" of the cell at right
for r in range(HEIGHT):
    for c in range(WIDTH):
        if r!=0:
            s.add(T[r][c]==B[r-1][c])
        if r!=HEIGHT-1:
            s.add(B[r][c]==T[r+1][c])
        if c!=0:
            s.add(L[r][c]==R[r][c-1])
        if c!=WIDTH-1:
            s.add(R[r][c]==L[r][c+1])

# "left" of each cell of first column shouldn't have any connection
# so is "right" of each cell of the last column
for r in range(HEIGHT):
    s.add(L[r][0]==f)
    s.add(R[r][WIDTH-1]==f)
```

```
# "top" of each cell of the first row shouldn't have any connection
# so is "bottom" of each cell of the last row
for c in range(WIDTH):
    s.add(T[0][c]==f)
    s.add(B[HEIGHT-1][c]==f)
```

Now we'll enumerate all cells in the initial array (8.3.1). First two cells are empty there. And the third one has type "2b". This is "┏" and it can be oriented in 4 possible ways. And if it has angle 0°, bottom and right half-joints are present, others are absent. If it has angle 90°, it looks like "┓", and bottom and left half-joints are present, others are absent.

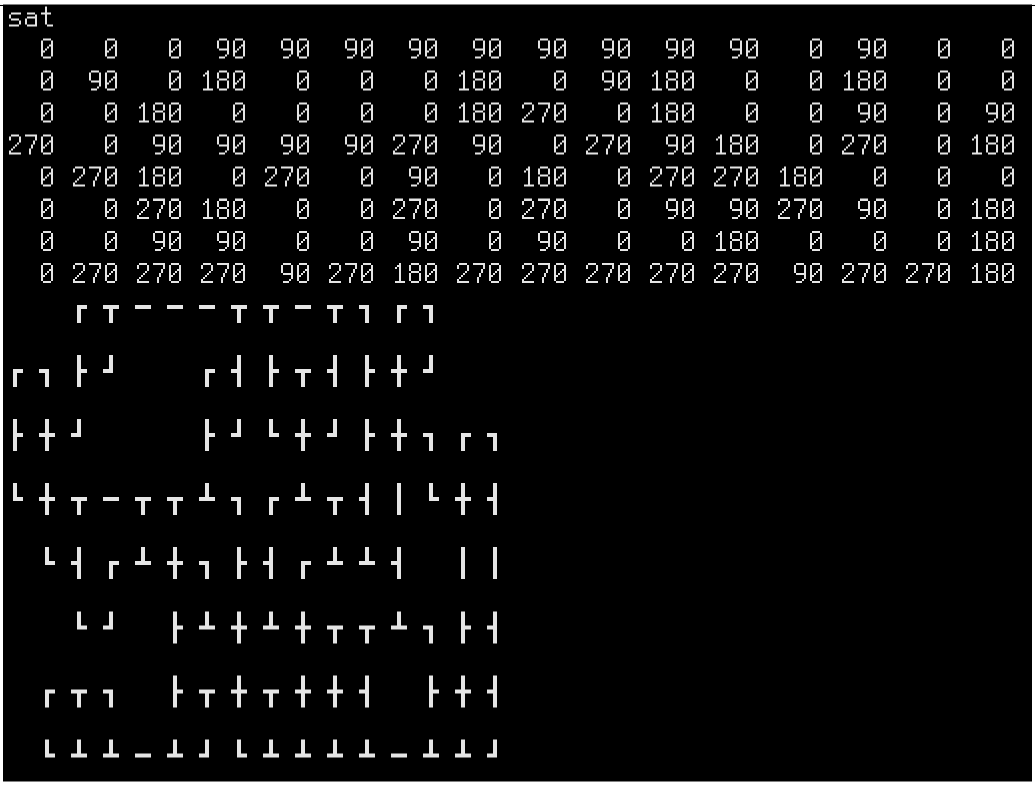
In plain English: "if cell of this type has angle 0°, these half-joints must be present **OR** if it has angle 90°, these half-joints must be present, **OR**, etc, etc."

Likewise, we define all these rules for all types and all possible angles:

```
for r in range(HEIGHT):
    for c in range(WIDTH):
        ty=cells_type[r][c]

        if ty=="0":
            s.add(A[r][c]==f)
            s.add(T[r][c]==f, B[r][c]==f, L[r][c]==f, R[r][c]==f)

        if ty=="2a":
            s.add(Or(And(A[r][c]==0, L[r][c]==f, R[r][c]==f, T[r][c]==t, B[r][c]==t),   # ┃
                    And(A[r][c]==90, L[r][c]==t, R[r][c]==t, T[r][c]==f, B[r][c]==f)))  # ━

        if ty=="2b":
            s.add(Or(And(A[r][c]==0, L[r][c]==f, R[r][c]==t, T[r][c]==f, B[r][c]==t),   # ┏
                    And(A[r][c]==90, L[r][c]==t, R[r][c]==f, T[r][c]==f, B[r][c]==t),   # ┓
                    And(A[r][c]==180, L[r][c]==t, R[r][c]==f, T[r][c]==t, B[r][c]==f),  # ┛
                    And(A[r][c]==270, L[r][c]==f, R[r][c]==t, T[r][c]==t, B[r][c]==f))) # ┗

        if ty=="3":
            s.add(Or(And(A[r][c]==0, L[r][c]==f, R[r][c]==t, T[r][c]==t, B[r][c]==t),   # ┣
                    And(A[r][c]==90, L[r][c]==t, R[r][c]==t, T[r][c]==f, B[r][c]==t),   # ┳
                    And(A[r][c]==180, L[r][c]==t, R[r][c]==f, T[r][c]==t, B[r][c]==t),  # ┫
                    And(A[r][c]==270, L[r][c]==t, R[r][c]==t, T[r][c]==t, B[r][c]==f))) # ┻

        if ty=="4":
            s.add(A[r][c]==0)
            s.add(T[r][c]==t, B[r][c]==t, L[r][c]==t, R[r][c]==t) # ╋
```

Full source code is here: https://smt.st/current_tree/puzzles/pipe/solver/solve_pipe_puzzle1.py.

It produces this result (prints angle for each cell and (pseudo)graphical representation):



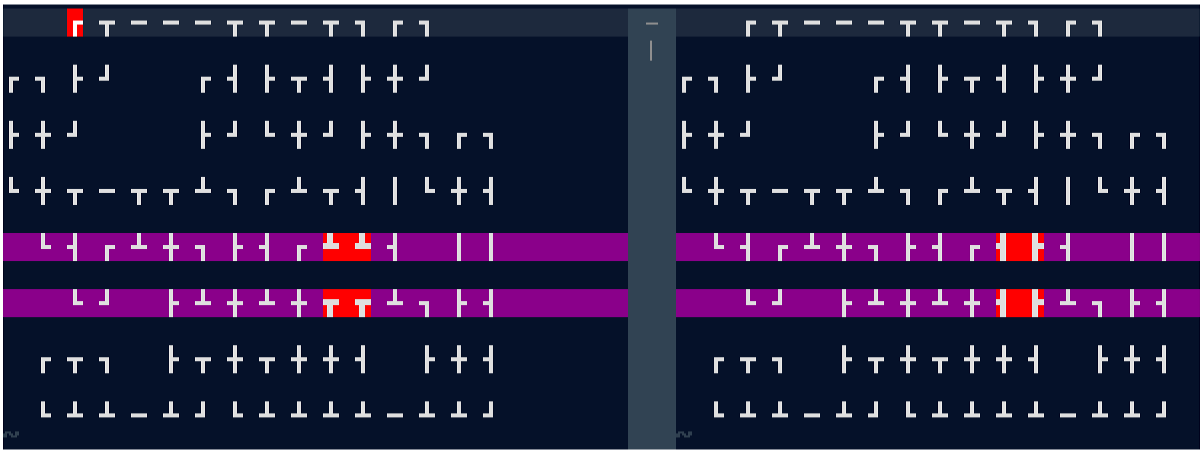
```
sat
  0   0   0  90  90  90  90  90  90  90  90  90   0  90   0   0
  0  90   0 180   0   0   0 180   0  90 180   0   0 180   0   0
  0   0 180   0   0   0   0 180 270   0 180   0   0  90   0  90
270   0  90  90  90  90 270  90   0 270  90 180   0 270   0 180
  0 270 180   0 270   0  90   0 180   0 270 270 180   0   0   0
  0   0 270 180   0   0 270   0 270   0  90  90 270  90   0 180
  0   0  90  90   0   0  90   0  90   0   0 180   0   0   0 180
  0 270 270 270  90 270 180 270 270 270 270 270  90 270 270 180
    ┌ ┬ ─ ─ ─ ┬ ┬ ─ ┬ ┐ ┌ ┐
┌ ┐ ├ ┘     ┌ ┤ ├ ┬ ┤ ├ ┼ ┘
├ ┼ ┘       ├ ┘ └ ┼ ┘ ├ ┼ ┐ ┌ ┐
└ ┼ ┬ ─ ┬ ┬ ┴ ┐ ┌ ┴ ┬ ┤ │ └ ┼ ┤
  └ ┤ ┌ ┴ ┼ ┐ ├ ┤ ┌ ┴ ┴ ┤   │ │
    └ ┘   ├ ┴ ┼ ┴ ┼ ┬ ┬ ┴ ┐ ├ ┤
  ┌ ┬ ┐   ├ ┬ ┼ ┬ ┼ ┼ ┤   ├ ┼ ┤
  └ ┴ ┴ ─ ┴ ┘ └ ┴ ┴ ┴ ┴ ─ ┴ ┴ ┘
```

Figure 8.5: Solver script output

It worked ≈ 4 seconds on my old and slow Intel Atom N455 1.66GHz. Is it fast? I don't know, but again, what is really cool, we do not know about any mathematical background of all this, we just defined cells, (half-)joints and defined relations between them.

Now the next question is, how many solutions are possible? Using method described earlier (3.17), I've altered solver script [7] and solver said two solutions are possible.

Let's compare these two solutions using gvimdiff:



```
    ┌ ┬ ─ ─ ─ ┬ ┬ ─ ┬ ┐ ┌ ┐
┌ ┐ ├ ┘     ┌ ┤ ├ ┬ ┤ ├ ┼ ┘
├ ┼ ┘       ├ ┘ └ ┼ ┘ ├ ┼ ┐ ┌ ┐
└ ┼ ┬ ─ ┬ ┬ ┴ ┐ ┌ ┴ ┬ ┤ │ └ ┼ ┤
  └ ┤ ┌ ┴ ┼ ┐ ├ ┤ ┌ ┴ ┴ ┤   │ │
    └ ┘   ├ ┴ ┼ ┴ ┼ ┬ ┬ ┴ ┐ ├ ┤
  ┌ ┬ ┐   ├ ┬ ┼ ┬ ┼ ┼ ┤   ├ ┼ ┤
  └ ┴ ┴ ─ ┴ ┘ └ ┴ ┴ ┴ ┴ ─ ┴ ┴ ┘
```

```
    ┌ ┬ ─ ─ ─ ┬ ┬ ─ ┬ ┐ ┌ ┐
┌ ┐ ├ ┘     ┌ ┤ ├ ┬ ┤ ├ ┼ ┘
├ ┼ ┘       ├ ┘ └ ┼ ┘ ├ ┼ ┐ ┌ ┐
└ ┼ ┬ ─ ┬ ┬ ┴ ┐ ┌ ┴ ┬ ┤ │ └ ┼ ┤
  └ ┤ ┌ ┴ ┼ ┐ ├ ┤ ┌ ┤ ├ ┤   │ │
    └ ┘   ├ ┴ ┼ ┴ ┼ ┤ ├ ┴ ┐ ├ ┤
  ┌ ┬ ┐   ├ ┬ ┼ ┬ ┼ ┼ ┤   ├ ┼ ┤
  └ ┴ ┴ ─ ┴ ┘ └ ┴ ┴ ┴ ┴ ─ ┴ ┴ ┘
```

Figure 8.6: gvimdiff output (pardon my red cursor at left pane at left-top corner)

4 cells in the middle can be orientated differently. Perhaps, other puzzles may produce different results.

P.S. *Half-joint* is defined as boolean type. But in fact, the first version of the solver has been written using integer type for half-joints, and 0 was used for False and 1 for True. I did it so because I wanted to make source code tidier and narrower without using long words like "False" and "True". And it worked, but slower. Perhaps, Z3 handles boolean data types faster? Better? Anyway, I writing this to note that integer type can also be used instead of boolean, if needed.

[7] https://smt.st/current_tree/puzzles/pipe/solver/solve_pipe_puzzle2.py

8.4 Eight queens problem (SAT)

Eight queens is a very popular problem and often used for measuring performance of SAT solvers. The problem is to place 8 queens on chess board so they will not attack each other. For example:

```
| | | |*| | | | |
| | | | | | |*| |
| | | | |*| | | |
| |*| | | | | | |
| | | | | |*| | |
|*| | | | | | | |
| | |*| | | | | |
| | | | | | | |*|
```

Let's try to figure out how to solve it.

8.4.1 make_one_hot

One important function we will (often) use is `make_one_hot`. This is a function which returns *True* if one single of inputs is *True* and others are *False*. It will return *False* otherwise.

In my other examples, I've used Wolfram Mathematica to generate CNF clauses for it, for example: 3.10.2. What expression Mathematica offers as `make_one_hot` function with 8 inputs?

```
(!a||!b)&&(!a||!c)&&(!a||!d)&&(!a||!e)&&(!a||!f)&&(!a||!g)&&(!a||!h)&&(a||b||c||d||e
    ||f||g||h)&&
(!b||!c)&&(!b||!d)&&(!b||!e)&&(!b||!f)&&(!b||!g)&&(!b||!h)&&(!c||!d)&&(!c||!e)&&(!c
    ||!f)&&(!c||!g)&&
(!c||!h)&&(!d||!e)&&(!d||!f)&&(!d||!g)&&(!d||!h)&&(!e||!f)&&(!e||!g)&&(!e||!h)&&(!f
    ||!g)&&(!f||!h)&&(!g||!h)
```

We can clearly see that the expression consisting of all possible variable pairs (negated) plus enumeration of all variables (non-negated). In plain English terms, this means: "no pair can be equal to two *True*'s *AND* at least one *True* must be present among all variables".

This is how it works: if two variables will be *True*, negated they will be both *False*, and this clause will not be evaluated to *True*, which is our ultimate goal. If one of variables is *True*, both negated will produce one *True* and one *False* (fine). If both variables are False, both negated will produce two *True's* (again, fine).

Here is how I can generate clauses for the function using *itertools* module from Python, which provides many important functions from combinatorics:

```
    # naive/pairwise encoding
    def AtMost1(self, lst):
        for pair in itertools.combinations(lst, r=2):
            self.add_clause([self.neg(pair[0]), self.neg(pair[1])])

    # make one-hot (AKA unitary) variable
    def make_one_hot(self, lst):
        self.AtMost1(lst)
        self.OR_always(lst)
```

`AtMost1()` function enumerates all possible pairs using *itertools* function *combinations()*.

`make_one_hot()` function does the same, but also adds a final clause, which forces at least one *True* variable to be present.

What clauses will be generated for 5 variables (1..5)?

```
p cnf 5 11
-2 -5 0
-2 -4 0
-4 -5 0
-2 -3 0
-1 -4 0
-1 -5 0
-1 -2 0
-1 -3 0
-3 -4 0
```

```
-3 -5 0
1 2 3 4 5 0
```

Yes, these are all possible pairs of 1..5 numbers + all 5 numbers.

We can get all solutions using Picosat:

```
% picosat --all popcnt1.cnf

s SATISFIABLE
v -1 -2 -3 -4 5 0
s SATISFIABLE
v -1 -2 -3 4 -5 0
s SATISFIABLE
v -1 -2 3 -4 -5 0
s SATISFIABLE
v -1 2 -3 -4 -5 0
s SATISFIABLE
v 1 -2 -3 -4 -5 0
s SOLUTIONS 5
```

5 possible solutions indeed.

8.4.2 Eight queens

Now let's get back to eight queens.

We can assign 64 variables to $8 \cdot 8 = 64$ cells. Cell occupied with queen will be *True*, vacant cell will be *False*.

The problem of placing non-attacking (each other) queens on chess board (of any size) can be stated in plain English like this:

- one single queen must be present at each row;
- one single queen must be present at each column;
- zero or one queen must be present at each diagonal (empty diagonals can be present in valid solution).

These rules can be translated like that:

- make_one_hot(each row)==*True*
- make_one_hot(each column)==*True*
- AtMost1(each diagonal)==*True*

Now all we need is to enumerate rows, columns and diagonals and gather all clauses:

```
#!/usr/bin/env python3

#-*- coding: utf-8 -*-

import itertools, subprocess, os, my_utils, SAT_lib

SIZE=8
SKIP_SYMMETRIES=True
#SKIP_SYMMETRIES=False

def row_col_to_var(row, col):
    global first_var
    return row*SIZE+col+first_var

def gen_diagonal(s, start_row, start_col, increment, limit):
    col=start_col
    tmp=[]
    for row in range(start_row, SIZE):
        tmp.append(row_col_to_var(row, col))
        col=col+increment
        if col==limit:
            break
    # ignore diagonals consisting of 1 cell:
    if len(tmp)>1:
        # we can't use POPCNT1() here, since some diagonals are empty in valid solutions.
        s.AtMost1(tmp)

def add_2D_array_as_negated_constraint(s, a):
    negated_solution=[]
    for row in range(SIZE):
        for col in range(SIZE):
            negated_solution.append(s.neg_if(a[row][col], row_col_to_var(row, col)))
    s.add_clause(negated_solution)

def main():
    global first_var

    s=SAT_lib.SAT_lib(False)

    _vars=s.alloc_BV(SIZE**2)
    first_var=_vars[0]

    # enumerate all rows:
    for row in range(SIZE):
        s.make_one_hot([row_col_to_var(row, col) for col in range(SIZE)])

    # enumerate all columns:
    # make_one_hot() could be used here as well:
    for col in range(SIZE):
        s.AtMost1([row_col_to_var(row, col) for row in range(SIZE)])

    # enumerate all diagonals:
    for row in range(SIZE):
        for col in range(SIZE):
            gen_diagonal(s, row, col, 1, SIZE) # from L to R
            gen_diagonal(s, row, col, -1, -1)  # from R to L

    # find all solutions:
    sol_n=1
    while True:
        if s.solve()==False:
            print ("unsat!")
            print ("solutions total=", sol_n-1)
            exit(0)

        # print solution:
        print ("solution number", sol_n, ":")

        # get solution and make 2D array of bools:
        solution_as_2D_bool_array=[]
        for row in range(SIZE):
            solution_as_2D_bool_array.append ([s.get_var_from_solution(row_col_to_var(row, col)) for col in range(SIZE)])

        # print 2D array:
        for row in range(SIZE):
            tmp=[([" ", "*"][solution_as_2D_bool_array[row][col]]+"|") for col in range(SIZE)]
            print ("|"+"".join(tmp))
        # add 2D array as negated constraint:
        add_2D_array_as_negated_constraint(s, solution_as_2D_bool_array)

        # if we skip symmetries, rotate/reflect soluion and add them as negated
            constraints:
        if SKIP_SYMMETRIES:
            for a in range(4):
                tmp=my_utils.rotate_rect_array(solution_as_2D_bool_array, a)
                add_2D_array_as_negated_constraint(s, tmp)

                tmp=my_utils.reflect_horizontally(my_utils.rotate_rect_array(
                    solution_as_2D_bool_array, a))
                add_2D_array_as_negated_constraint(s, tmp)

        sol_n=sol_n+1

main()
```

(https://smt.st/current_tree/puzzles/8queens/8queens.py)

Perhaps, gen_diagonal() function is not very aesthetically appealing: it enumerates also subdiagonals of already enumerated longer diagonals. To prevent presence of duplicate clauses, *clauses* global variable is not a list, rather set, which allows only unique data to be present there.

Also, I've used AtMost1 for each column, this will help to produce slightly lower number of clauses. Each column will have a queen anyway, this is implied from the first rule (make_one_hot for each row).

After running, we got CNF file with 64 variables and 736 clauses (https://smt.st/current_tree/puzzles/8queens/8queens.cnf). Here is one solution:

```
% python 8queens.py
len(clauses)= 736
| | | |*| | | | |
| | | | | | |*| |
| | | | |*| | | |
| |*| | | | | | |
| | | | | |*| | |
|*| | | | | | | |
| | |*| | | | | |
| | | | | | | |*|
```

How many possible solutions are there? Picosat tells 92, which is indeed correct number of solutions (https://oeis.org/A000170).

Performance of Picosat is not impressive, probably because it has to output all the solutions. It took 34 seconds on my ancient Intel Atom 1.66GHz netbook to enumerate all solutions for $11 \cdot 11$ chess board (2680 solutions), which is way slower than my straight brute-force program: https://yurichev.com/blog/8queens/. Nevertheless, it's lighting fast (as other SAT solvers) in finding first solution.

The SAT instance is also small enough to be easily solved by my simplest possible backtracking SAT solver: 23.1.

8.4.3 Counting all solutions

We get a solution, negate it and add as a new constraint. In plain English language this sounds "find a solution, which is also can't be equal to the recently found/added". We add them consequently and the process is slowing—just because a problem (*instance*) is growing and SAT solver experience hard times in find yet another solution.

8.4.4 Skipping symmetrical solutions

We can also add rotated and reflected (horizontally) solution, so to skip symmetrical solutions. By doing so, we're getting 12 solutions for 8*8 board, 46 for 9*9 board, etc. This is https://oeis.org/A002562.

8.5 Solving pocket Rubik's cube (2*2*2) using Z3

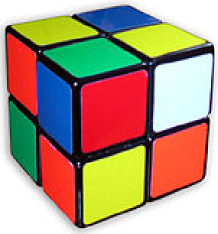


Figure 8.7: Pocket cube

(The image has been taken from Wikipedia.)

Solving Rubik's cube is not a problem, finding shortest solution is.

8.5.1 Intro

First, a bit of terminology. There are 6 colors we have: white, green, blue, orange, red, yellow. We also have 6 sides: front, up, down, left, right, back.

This is how we will name all facelets:

```
        U1 U2
        U3 U4

       -------
L1 L2 | F1 F2 | R1 R2 | B1 B2
L3 L4 | F3 F4 | R3 R4 | B3 B4
       -------

        D1 D2
        D3 D4
```

Colors on a solved cube are:

```
    G G
    G G
    ---
R R|W W|O O|Y Y
R R|W W|O O|Y Y
    ---
    B B
    B B
```

There are 6 possible turns: front, left, right, back, up, down. But each turn can be clockwise, counterclockwise and half-turn (equal to two CW or two CCW). Each CW is equal to 3 CCW and vice versa. Hence, there are 6*3=18 possible turns.

It is known, that 11 turns (including half-turns) are enough to solve any pocket cube (God's algorithm). This means, graph has a diameter of 11. For 3*3*3 cube one need 20 turns (http://www.cube20.org/). See also: https://en.wikipedia.org/wiki/Rubik%27s_Cube_group.

8.5.2 Z3

There are 6 sides and 4 facelets on each, hence, 6*4=24 variables we need to define a state.

Then we define how state is transformed after each possible turn:

```
FACE_F, FACE_U, FACE_D, FACE_R, FACE_L, FACE_B = 0,1,2,3,4,5

def rotate_FCW(s):
    return [
        [ s[FACE_F][2], s[FACE_F][0], s[FACE_F][3], s[FACE_F][1] ],   # for F
        [ s[FACE_U][0], s[FACE_U][1], s[FACE_L][3], s[FACE_L][1] ],   # for U
        [ s[FACE_R][2], s[FACE_R][0], s[FACE_D][2], s[FACE_D][3] ],   # for D
        [ s[FACE_U][2], s[FACE_R][1], s[FACE_U][3], s[FACE_R][3] ],   # for R
        [ s[FACE_L][0], s[FACE_D][0], s[FACE_L][2], s[FACE_D][1] ],   # for L
        [ s[FACE_B][0], s[FACE_B][1], s[FACE_B][2], s[FACE_B][3] ] ]  # for B

def rotate_FH(s):
    return [
        [ s[FACE_F][3], s[FACE_F][2], s[FACE_F][1], s[FACE_F][0] ],
        [ s[FACE_U][0], s[FACE_U][1], s[FACE_D][1], s[FACE_D][0] ],
        [ s[FACE_U][3], s[FACE_U][2], s[FACE_D][2], s[FACE_D][3] ],
        [ s[FACE_L][3], s[FACE_R][1], s[FACE_L][1], s[FACE_R][3] ],
        [ s[FACE_L][0], s[FACE_R][2], s[FACE_L][2], s[FACE_R][0] ],
        [ s[FACE_B][0], s[FACE_B][1], s[FACE_B][2], s[FACE_B][3] ] ]

...
```

Then we define a function, which takes turn number and transforms a state:

```
# op is turn number
def rotate(turn, state, face, facelet):
    return If(op==0,  rotate_FCW (state)[face][facelet],
           If(op==1,  rotate_FCCW(state)[face][facelet],
           If(op==2,  rotate_UCW (state)[face][facelet],
           If(op==3,  rotate_UCCW(state)[face][facelet],
           If(op==4,  rotate_DCW (state)[face][facelet],

...

           If(op==17, rotate_BH  (state)[face][facelet],
                      0))))))))))))))))))
```

Now set "solved" state, initial state and connect everything:

```
move_names=["FCW", "FCCW", "UCW", "UCCW", "DCW", "DCCW", "RCW", "RCCW", "LCW", "LCCW
    ", "BCW", "BCCW", "FH", "UH", "DH", "RH", "LH", "BH"]

def colors_to_array_of_ints(s):
    return [{"W":0, "G":1, "B":2, "O":3, "R":4, "Y":5}[c] for c in s]

def set_current_state (d):
    F=colors_to_array_of_ints(d["FACE_F"])
    U=colors_to_array_of_ints(d["FACE_U"])
    D=colors_to_array_of_ints(d["FACE_D"])
    R=colors_to_array_of_ints(d["FACE_R"])
    L=colors_to_array_of_ints(d["FACE_L"])
    B=colors_to_array_of_ints(d["FACE_B"])
    return F,U,D,R,L,B # return tuple

# 4
init_F, init_U, init_D, init_R, init_L, init_B=set_current_state({"FACE_F":"RYOG", "
    FACE_U":"YRGO", "FACE_D":"WRBO", "FACE_R":"GYWB", "FACE_L":"BYWG", "FACE_B":"BOWR
    "})

...

for TURNS in range(1,12): # 1..11
    print "turns=", TURNS

    s=Solver()

    state=[[[Int('state%d_%d_%d' % (n, side, i)) for i in range(FACELETS)] for side
        in range(FACES)] for n in range(TURNS+1)]
    op=[Int('op%d' % n) for n in range(TURNS+1)]

    for i in range(FACELETS):
        s.add(state[0][FACE_F][i]==init_F[i])
        s.add(state[0][FACE_U][i]==init_U[i])
        s.add(state[0][FACE_D][i]==init_D[i])
        s.add(state[0][FACE_R][i]==init_R[i])
        s.add(state[0][FACE_L][i]==init_L[i])
        s.add(state[0][FACE_B][i]==init_B[i])

    # solved state
    for face in range(FACES):
        for facelet in range(FACELETS):
            s.add(state[TURNS][face][facelet]==face)

    # turns:
    for turn in range(TURNS):
        for face in range(FACES):
            for facelet in range(FACELETS):
                s.add(state[turn+1][face][facelet]==rotate(op[turn], state[turn],
                   face, facelet))

    if s.check()==sat:
        print "sat"
        m=s.model()
        for turn in range(TURNS):
            print move_names[int(str(m[op[turn]]))]
        exit(0)
```

(The full source code: https://smt.st/current_tree/puzzles/rubik2/failed_SMT/rubik2_z3.py)

That works:

```
turns= 1
turns= 2
turns= 3
turns= 4
sat
RCW
UCW
DCW
RCW
```

...but very slow. It takes up to 1 hours to find a path of 8 turns, which is not enough, we need 11.

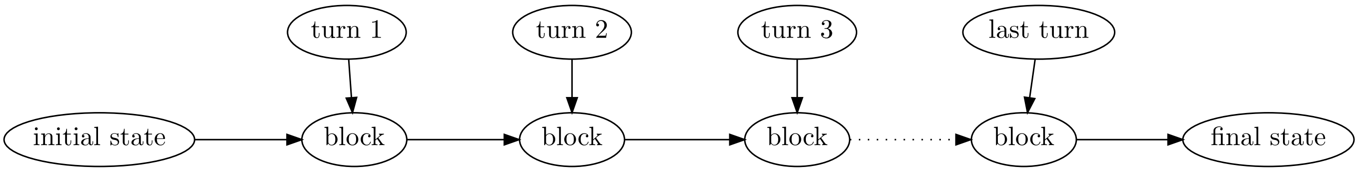
Nevertheless, I decided to include Z3 solver as a demonstration.

8.6 Pocket Rubik's Cube (2*2*2) and SAT solver

.

I had success with my SAT-based solver, which can find an 11-turn path for a matter of 10-20 minutes on my old Intel Xeon E3-1220 3.10GHz.

First, we will encode each color as 3-bit bit vector. Then we can build electronic circuit, which will take initial state of cube and output final state. It can have switches for each turn on each state.



You set all turns and the device ”calculates” final state.

Each "blk" can be consisted of 24 multiplexers (MUX), each for each facelet. Each MUX is controlled by 5-bit command (turn number). Depending on command, MUX takes 3-bit color from a facelet from a previous state.

Here is a table: the first column is a "destination" facelet, then a list of "source" facelets for each turn:

```
#      dst,  FCW  FH   FCCW UCW  UH   UCCW DCW  DH   DCCW RCW  RH   RCCW LCW  LH
       LCCW BCW  BH   BCCW
add_r("F1",["F3","F4","F2","R1","B1","L1","F1","F1","F1","F1","F1","F1","U1","B4
    ","D1","F1","F1","F1"])
add_r("F2",["F1","F3","F4","R2","B2","L2","F2","F2","F2","D2","B3","U2","F2","F2
    ","F2","F2","F2","F2"])
add_r("F3",["F4","F2","F1","F3","F3","F3","L3","B3","R3","F3","F3","F3","U3","B2
    ","D3","F3","F3","F3"])
add_r("F4",["F2","F1","F3","F4","F4","F4","L4","B4","R4","D4","B1","U4","F4","F4
    ","F4","F4","F4","F4"])
add_r("U1",["U1","U1","U1","U3","U4","U2","U1","U1","U1","U1","U1","U1","B4","D1
    ","F1","R2","D4","L3"])
add_r("U2",["U2","U2","U2","U1","U3","U4","U2","U2","U2","F2","D2","B3","U2","U2
    ","U2","R4","D3","L1"])
add_r("U3",["L4","D2","R1","U4","U2","U1","U3","U3","U3","U3","U3","U3","B2","D3
    ","F3","U3","U3","U3"])
add_r("U4",["L2","D1","R3","U2","U1","U3","U4","U4","U4","F4","D4","B1","U4","U4
    ","U4","U4","U4","U4"])
add_r("D1",["R3","U4","L2","D1","D1","D1","D3","D4","D2","D1","D1","D1","F1","U1
    ","B4","D1","D1","D1"])
add_r("D2",["R1","U3","L4","D2","D2","D2","D1","D3","D4","B3","U2","F2","D2","D2
    ","D2","D2","D2","D2"])
add_r("D3",["D3","D3","D3","D3","D3","D3","D4","D2","D1","D3","D3","D3","F3","U3
    ","B2","L1","U2","R4"])
add_r("D4",["D4","D4","D4","D4","D4","D4","D2","D1","D3","B1","U4","F4","D4","D4
    ","D4","L3","U1","R2"])
add_r("R1",["U3","L4","D2","B1","L1","F1","R1","R1","R1","R3","R4","R2","R1","R1
    ","R1","R1","R1","R1"])
add_r("R2",["R2","R2","R2","B2","L2","F2","R2","R2","R2","R1","R3","R4","R2","R2
    ","R2","D4","L3","U1"])
add_r("R3",["U4","L2","D1","R3","R3","R3","F3","L3","B3","R4","R2","R1","R3","R3
    ","R3","R3","R3","R3"])
add_r("R4",["R4","R4","R4","R4","R4","R4","F4","L4","B4","R2","R1","R3","R4","R4
    ","R4","D3","L1","U2"])
add_r("L1",["L1","L1","L1","F1","R1","B1","L1","L1","L1","L1","L1","L1","L3","L4
    ","L2","U2","R4","D3"])
add_r("L2",["D1","R3","U4","F2","R2","B2","L2","L2","L2","L2","L2","L2","L1","L3
    ","L4","L2","L2","L2"])
add_r("L3",["L3","L3","L3","L3","L3","L3","B3","R3","F3","L3","L3","L3","L4","L2
    ","L1","U1","R2","D4"])
add_r("L4",["D2","R1","U3","L4","L4","L4","B4","R4","F4","L4","L4","L4","L2","L1
    ","L3","L4","L4","L4"])
add_r("B1",["B1","B1","B1","L1","F1","R1","B1","B1","B1","U4","F4","D4","B1","B1
    ","B1","B3","B4","B2"])
add_r("B2",["B2","B2","B2","L2","F2","R2","B2","B2","B2","B2","B2","B2","D3","F3
    ","U3","B1","B3","B4"])
add_r("B3",["B3","B3","B3","B3","B3","B3","R3","F3","L3","U2","F2","D2","B3","B3
    ","B3","B4","B2","B1"])
add_r("B4",["B4","B4","B4","B4","B4","B4","R4","F4","L4","B4","B4","B4","D1","F1
    ","U1","B2","B1","B3"])
```

Each MUX has 32 inputs, each has width of 3 bits: colors from "source" facelets. It has 3-bit output (color for "destination" facelet). It has 5-bit selector, for 18 turns. Other selector values (32-18=14 values) are not used at all.

The whole problem is to build a circuit and then ask SAT solver to set "switches" to such a state, when input and output are determined (by us).

Now the problem is to represent MUX in CNF terms.

From EE[8] courses we can remember about a simple if-then-else (ITE) gate, it takes 3 inputs ("selector", "true" and "false") and it has 1 output. Depending on "selector" input it "connects" output with "true" or "false" input. Using tree of ITE gates we first can build 32-to-1 MUX, then wide 32*3-to-3 MUX.

I once have written small utility to search for shortest possible CNF formula for a specific function, in a bruteforce manner (https://smt.st/current_tree/puzzles/rubik2/SAT/XOR_CNF_bf.c). It was inspired by "aha! hacker assistant" by Henry Warren. So here is a function:

```
bool func(bool v[VARIABLES])
{

        // ITE:

        bool tmp;
        if (v[0]==0)
                tmp=v[1];
        else
                tmp=v[2];

        return tmp==v[3];
}
```

A shortest CNF for it:

```
try_all_CNFs_of_len(1)
try_all_CNFs_of_len(2)
try_all_CNFs_of_len(3)
try_all_CNFs_of_len(4)
found a CNF:
p cnf 4 4
-1 3 -4 0
1 2 -4 0
-1 -3 4 0
1 -2 4 0
```

1st variable is "select", 2nd is "false", 3rd is "true", 4th is "output". "output" is an additional variable, added just like in Tseitin transformations.

Hence, CNF formula is:

```
(!select OR true OR !output) AND (select OR false OR !output) AND (!select OR !true
   OR output) AND (select OR !false OR output)
```

It assures that the "output" will always be equal to one of inputs depending on "select".

Now we can build a tree of ITE gates:

```
def create_ITE(s,f,t):
    x=create_var()

    clauses.append([neg(s),t,neg(x)])
    clauses.append([s,f,neg(x)])
    clauses.append([neg(s),neg(t),x])
    clauses.append([s,neg(f),x])

    return x

# ins=16 bits
# sel=4 bits
def create_MUX(ins, sel):
    t0=create_ITE(sel[0],ins[0],ins[1])
    t1=create_ITE(sel[0],ins[2],ins[3])
    t2=create_ITE(sel[0],ins[4],ins[5])
    t3=create_ITE(sel[0],ins[6],ins[7])
    t4=create_ITE(sel[0],ins[8],ins[9])
```

[8] Electrical engineering

```
    t5=create_ITE(sel[0],ins[10],ins[11])
    t6=create_ITE(sel[0],ins[12],ins[13])
    t7=create_ITE(sel[0],ins[14],ins[15])

    y0=create_ITE(sel[1],t0,t1)
    y1=create_ITE(sel[1],t2,t3)
    y2=create_ITE(sel[1],t4,t5)
    y3=create_ITE(sel[1],t6,t7)

    z0=create_ITE(sel[2],y0,y1)
    z1=create_ITE(sel[2],y2,y3)

    return create_ITE(sel[3], z0, z1)
```

This is my old MUX I wrote for 16 inputs and 4-bit selector, but you've got the idea: this is 4-tier tree. It has 15 ITE gates or 15*4=60 clauses.

Now the question, is it possible to make it smaller? I've tried to use Mathematica. First I've built truth table for 4-bit selector:

```
...
1 1 1 1     1 1 1 1 1 1 0 1 0 1 0 1 1 0 1 1    0    0
1 1 1 1     1 1 1 1 1 1 0 1 0 1 0 1 1 0 1 1    1    1
1 1 1 1     1 1 1 1 1 1 0 1 0 1 0 1 1 1 0 0    0    0
1 1 1 1     1 1 1 1 1 1 0 1 0 1 0 1 1 1 0 0    1    1
1 1 1 1     1 1 1 1 1 1 0 1 0 1 0 1 1 1 0 1    0    0
1 1 1 1     1 1 1 1 1 1 0 1 0 1 0 1 1 1 0 1    1    1
1 1 1 1     1 1 1 1 1 1 0 1 0 1 0 1 1 1 1 0    0    0
1 1 1 1     1 1 1 1 1 1 0 1 0 1 0 1 1 1 1 0    1    1
1 1 1 1     1 1 1 1 1 1 0 1 0 1 0 1 1 1 1 1    0    0
1 1 1 1     1 1 1 1 1 1 0 1 0 1 0 1 1 1 1 1    1    1
1 1 1 1     1 1 1 1 1 1 0 1 0 1 1 0 0 0 0 0    0    0
1 1 1 1     1 1 1 1 1 1 0 1 0 1 1 0 0 0 0 0    1    1
1 1 1 1     1 1 1 1 1 1 0 1 0 1 1 0 0 0 0 1    0    0
1 1 1 1     1 1 1 1 1 1 0 1 0 1 1 0 0 0 0 1    1    1
1 1 1 1     1 1 1 1 1 1 0 1 0 1 1 0 0 0 1 0    0    0
1 1 1 1     1 1 1 1 1 1 0 1 0 1 1 0 0 0 1 0    1    1
1 1 1 1     1 1 1 1 1 1 0 1 0 1 1 0 0 0 1 1    0    0
...
```

First 4 bits is selector, then 16 bits of input. Then the possible output and the bit, indicating, if the output bit equals to one of the inputs.

Then I load this table to Mathematica and make CNF expression out of truth table:

```
arr=Import["/home/dennis/P/Rubik/TT","Table"]
{{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0},
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0},
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1},
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,1},
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0},
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0},
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1},
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1},
...2097135... ,
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0},
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,1,1},
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,0,0},
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1},
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0},
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1},
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0},
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}}
```

```
TT=(Most[#]->Last[#])&/@arr
{{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}->1,
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1}->0,
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0}->0,
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1}->1,
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0}->1,
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1}->0,
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0}->0,
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1}->1,
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0}->1,
...2097135... ,
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0}->0,
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,1}->1,
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,0}->0,
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1}->1,
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0}->0,
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1}->1,
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0}->0,
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}->1}


BooleanConvert[BooleanFunction[TT,{s3,s2,s1,s0, i15,i14,i13,i12,i11,i10,i9,i8,i7,i6,
    i5,i4,i3, i2, i1, i0,x}],"CNF"]
(!i0||s0||s1||s2||s3||x)&&(i0||s0||s1||s2||s3||!x)&&(!i1||!s0||s1||s2||s3||x)&&(i1||!
    s0||s1||s2||s3||!x)&&(!i10||s0||!s1||s2||!s3||x)&&
(i10||s0||!s1||s2||!s3||!x)&&(!i11||!s0||!s1||s2||!s3||x)&&(i11||!s0||!s1||s2||!s3||!
    x)&&(!i12||s0||s1||!s2||!s3||x)&&(i12||s0||s1||!s2||!s3||!x)&&
(!i13||!s0||s1||!s2||!s3||x)&&(i13||!s0||s1||!s2||!s3||!x)&&(!i14||s0||!s1||!s2||!s3
    ||x)&&(i14||s0||!s1||!s2||!s3||!x)&&
(!i15||!s0||!s1||!s2||!s3||x)&&(i15||!s0||!s1||!s2||!s3||!x)&&(!i2||s0||!s1||s2||s3||
    x)&&(i2||s0||!s1||s2||s3||!x)&&(!i3||!s0||!s1||s2||s3||x)&&
(i3||!s0||!s1||s2||s3||!x)&&(!i4||s0||s1||!s2||s3||x)&&(i4||s0||s1||!s2||s3||!x)&&(!
    i5||!s0||s1||!s2||s3||x)&&(i5||!s0||s1||!s2||s3||!x)&&
(!i6||s0||!s1||!s2||s3||x)&&(i6||s0||!s1||!s2||s3||!x)&&(!i7||!s0||!s1||!s2||s3||x)
    &&(i7||!s0||!s1||!s2||s3||!x)&&(!i8||s0||s1||s2||!s3||x)&&
(i8||s0||s1||s2||!s3||!x)&&(!i9||!s0||s1||s2||!s3||x)&&(i9||!s0||s1||s2||!s3||!x)
```

Look closer to CNF expression:

```
(!i0|    |s0||s1||s2|| s3|| x)&&
( i0||  s0|| s1||s2|| s3||!x)&&
(!i1||! s0|| s1||s2|| s3|| x)&&
( i1||! s0|| s1||s2|| s3||!x)&&
(!i10|| s0||!s1||s2||!s3|| x)&&
( i10|| s0||!s1||s2||!s3||!x)&&
(!i11||!s0||!s1||s2||!s3|| x)&&
( i11||!s0||!s1||s2||!s3||!x)&&

...

(!i8|| s0||s1||s2||!s3|| x)&&
( i8|| s0||s1||s2||!s3||!x)&&
(!i9||!s0||s1||s2||!s3|| x)&&
( i9||!s0||s1||s2||!s3||!x)
```

It has simple structure and there are 32 clauses, against 60 in my previous attempt. Will it work faster? No, as my experience shows, it doesn't speed up anything. Anyway, I used the latter idea to make MUX.

The following function makes pack of MUXes for each state, based on what I've got from Mathematica:

```
    def create_MUX(self, ins, sels):
        assert 2**len(sels)==len(ins)
        x=self.create_var()
        for sel in range(len(ins)): # 32 for 5-bit selector
            tmp=[self.neg_if((sel>>i)&1==1, sels[i]) for i in range(len(sels))] # 5 for 5-bit selector

            self.add_clause([self.neg(ins[sel])] + tmp + [x])
            self.add_clause([ins[sel]] + tmp + [self.neg(x)])
        return x

    def create_wide_MUX (self, ins, sels):
        out=[]
        for i in range(len(ins[0])):
            inputs=[x[i] for x in ins]
            out.append(self.create_MUX(inputs, sels))
        return out
```

Now the function that glues all together:

```
# src=list of 18 facelets
def add_r(dst, src):
    global facelets, selectors, s
    t=s.create_var()
    s.fix(t,True)
    for state in range(STATES-1):
        src_vectors=[]
        for tmp in src:
            src_vectors.append(facelets[state][tmp])

        # padding: there are 18 used MUX inputs, so add 32-18=14 for padding
        for i in range(32-18):
            src_vectors.append([t,t,t])

        dst_vector=s.create_wide_MUX (src_vectors, selectors[state])
        s.fix_BV_EQ(dst_vector,facelets[state+1][dst])

...

    #      dst,  FCW   FH    FCCW UCW   UH    UCCW  DCW   DH    DCCW  RCW   RH    RCCW  LCW   LH    LCCW  BCW   BH    BCCW
    add_r("F1",["F3","F4","F2","R1","B1","L1","F1","F1","F1","F1","F1","F1","U1","B4","D1","F1","F1","F1"])
    add_r("F2",["F1","F3","F4","R2","B2","L2","F2","F2","F2","D2","B3","U2","F2","F2","F2","F2","F2","F2"])
    add_r("F3",["F4","F2","F1","F3","F3","F3","L3","B3","R3","F3","F3","F3","U3","B2","D3","F3","F3","F3"])
    add_r("F4",["F2","F1","F3","F4","F4","F4","L4","B4","R4","D4","B1","U4","F4","F4","F4","F4","F4","F4"])
    add_r("U1",["U1","U1","U1","U3","U4","U2","U1","U1","U1","U1","U1","U1","B4","D1","F1","R2","D4","L3"])
...
```

Now the full source code: https://smt.st/current_tree/puzzles/rubik2/SAT/solver.py. I tried to make it as concise as possible. It requires minisat to be installed.

And it works up to 11 turns, starting at 11, then decreasing number of turns. Here is an output for a state which can be solved with 4 turns:

```
set_state(0, {"F":"RYOG", "U":"YRGO", "D":"WRBO", "R":"GYWB", "L":"BYWG", "B":"BOWR"})
```

```
TURNS= 11
sat!
RCW
DH
```

```
BCW
UCW
UH
DCCW
FH
BH
BH
FH
UH

TURNS= 10
sat!
RCCW
UCCW
UCW
RCW
RCW
UH
FCW
UCCW
DCCW
DH

TURNS= 9
sat!
BCCW
LH
RH
BCW
RCW
RCCW
DCW
FH
LCW

TURNS= 8
sat!
RCW
UH
BCW
UCCW
LCW
LH
RCW
FCW

TURNS= 7
sat!
DCCW
DH
UH
UCW
BCW
UH
RCW

TURNS= 6
sat!
RCW
UCCW
DCCW
```

```
LCW
UH
DH

TURNS= 5
sat!
LCW
FCW
BCW
RH
LCCW

TURNS= 4
sat!
RCW
DCW
UCW
RCW

TURNS= 3
unsat!
```

Even on my relic Intel Atom 1.5GHz netbook it takes just 20s.

You can find redundant turns in 11-turn solution, like double UH turns. Of course, two UH turns returns the cube to the previous state. So these ”annihilating turns” are added if the final solution can be shorter. Why the solver added it? There is no ”no operation” turn. And the solver is forced to fit into 11 turns. Hence, it do what it can to produce correct solution.

Now a hard example:

```
set_state(0, {"F":"RORW", "U":"BRBB", "D":"GOOR", "R":"WYGY", "L":"OWYW", "B":"BYGG
    "})
```

```
TURNS= 11
sat!
UCW
BCW
LCCW
BH
DCW
RH
DCCW
FCW
UH
LCW
RCW

TURNS= 10
sat!
RH
BCCW
LCCW
RH
BCCW
LH
BCW
DH
BCW
LCCW

...
```

(≈ 5 minutes on my old Intel Xeon E3-1220 3.10GHz.)

I couldn't find a pure "11-turn state" which is "unsat" for 10-turn, it seems, these are rare. According to wikipedia, there are just 0.072% of these states. Like "20-turn states" for 3*3*3 cube, which are also rare.

8.6.1 Several solutions

According to picosat (–all option to get all possible solutions), the 4-turn example we just saw has 2 solutions. Indeed, two consequent turns UCW and DCW can be interchanged, they do not conflict with each other.

8.6.2 Other (failed) ideas

Pocket cube (2*2*2) has no central facelets, so to solve it, you don't need to stick each color to each face. Rather, you can define a constraint so that a colors on each face must be equal to each other. Somehow, this slows down drastically my both Z3 and SAT-based solvers.

Also, to prevent "annihilating" turns, we can set a constraint so that each state cannot be equal to any of previous states, i.e., states cannot repeat. This also slows down my both solvers.

8.6.3 3*3*3 cube

3*3*3 cube requires much more turns (20), so I couldn't solve it with my methods. I have success to solve maybe 10 or 11 turns. But some people do all 20 turns: Jingchao Chen.

However, you can use 3*3*3 cube to play, because it can act as 2*2*2 cube: just use corners and ignore edge and center cubes. Here is mine I used, you can see that corners are correctly solved:

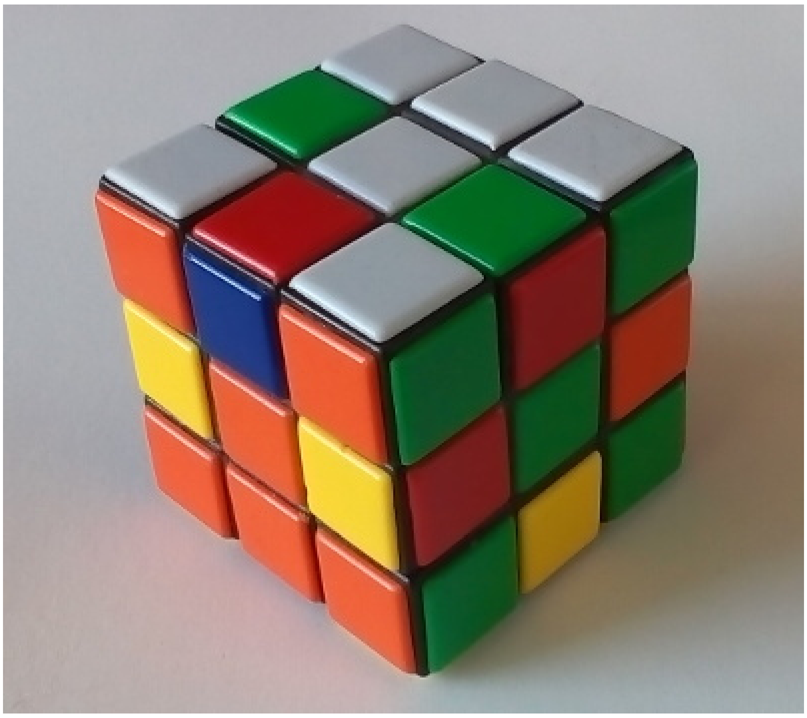


Figure 8.8: Solved Rubik's cube as a pocket cube

8.6.4 Some discussion

https://news.ycombinator.com/item?id=15214439,
https://www.reddit.com/r/compsci/comments/6zb34i/solving_pocket_rubiks_cube_222_using_z3_and_sat/,
https://www.reddit.com/r/Cubers/comments/6ze3ua/theory_dennis_yurichev_solving_pocket_rubiks_cube/.

8.7 Rubik's cube (3*3*3) and Z3 SMT-solver

As I wrote before, I couldn't solve even 2*2*2 pocket cube with Z3 (8.5), but I wanted to play with it further, and found that it's still possible to solve one face instead of all 6.

I tried to model color of each facelet using integer sort (type), but now I can use bool: white facelet is True and all other non-white is False. I can encode state of Rubik's cube like that: "W" for white facelet, "." for other.

Now the process of solving is a matter of minutes on a decent computer, or faster.

```
#!/usr/bin/env python

from z3 import *

FACES=6
FACELETS=9

def rotate_FCW(s):
        return [
                [ s[0][6], s[0][3], s[0][0], s[0][7], s[0][4], s[0][1], s[0][8], s
                   [0][5], s[0][2], ],  # new F
                [ s[1][0], s[1][1], s[1][2], s[1][3], s[1][4], s[1][5], s[4][8], s
                   [4][5], s[4][2], ],  # new U
                [ s[3][6], s[3][3], s[3][0], s[2][3], s[2][4], s[2][5], s[2][6], s
                   [2][7], s[2][8], ],  # new D
                [ s[1][6], s[3][1], s[3][2], s[1][7], s[3][4], s[3][5], s[1][8], s
                   [3][7], s[3][8], ],  # new R
                [ s[4][0], s[4][1], s[2][0], s[4][3], s[4][4], s[2][1], s[4][6], s
                   [4][7], s[2][2], ],  # new L
                [ s[5][0], s[5][1], s[5][2], s[5][3], s[5][4], s[5][5], s[5][6], s
                   [5][7], s[5][8], ] ] # new B

def rotate_FH(s):
        return [
                [ s[0][8], s[0][7], s[0][6], s[0][5], s[0][4], s[0][3], s[0][2], s
                   [0][1], s[0][0], ],
                [ s[1][0], s[1][1], s[1][2], s[1][3], s[1][4], s[1][5], s[2][2], s
                   [2][1], s[2][0], ],
                [ s[1][8], s[1][7], s[1][6], s[2][3], s[2][4], s[2][5], s[2][6], s
                   [2][7], s[2][8], ],
                [ s[4][8], s[3][1], s[3][2], s[4][5], s[3][4], s[3][5], s[4][2], s
                   [3][7], s[3][8], ],
                [ s[4][0], s[4][1], s[3][6], s[4][3], s[4][4], s[3][3], s[4][6], s
                   [4][7], s[3][0], ],
                [ s[5][0], s[5][1], s[5][2], s[5][3], s[5][4], s[5][5], s[5][6], s
                   [5][7], s[5][8], ] ]


...


def rotate(op, st, side, j):
        return If(op==0, rotate_FCW(st)[side][j],
                If(op==1, rotate_FCCW(st)[side][j],
                If(op==2, rotate_UCW(st)[side][j],
                If(op==3, rotate_UCCW(st)[side][j],
                If(op==4, rotate_DCW(st)[side][j],
                If(op==5, rotate_DCCW(st)[side][j],
                If(op==6, rotate_RCW(st)[side][j],
                If(op==7, rotate_RCCW(st)[side][j],
                If(op==8, rotate_LCW(st)[side][j],
                If(op==9, rotate_LCCW(st)[side][j],
                If(op==10, rotate_BCW(st)[side][j],
                If(op==11, rotate_BCCW(st)[side][j],
                If(op==12, rotate_FH(st)[side][j],
                If(op==13, rotate_UH(st)[side][j],
                If(op==14, rotate_DH(st)[side][j],
                If(op==15, rotate_RH(st)[side][j],
                If(op==16, rotate_LH(st)[side][j],
                If(op==17, rotate_BH(st)[side][j],
                        rotate_BH(st)[side][j], # default
                        ))))))))))))))))))

move_names=["FCW", "FCCW", "UCW", "UCCW", "DCW", "DCCW", "RCW", "RCCW", "LCW", "LCCW
   ", "BCW", "BCCW", "FH", "UH", "DH", "RH", "LH", "BH"]

def colors_to_array_of_ints(s):
    rt=[]
    for c in s:
        if c=='W':
            rt.append(True)
        else:
            rt.append(False)
    return rt

def set_current_state (d):
    F=colors_to_array_of_ints(d["F"])
    U=colors_to_array_of_ints(d["U"])
    D=colors_to_array_of_ints(d["D"])
    R=colors_to_array_of_ints(d["R"])
    L=colors_to_array_of_ints(d["L"])
    B=colors_to_array_of_ints(d["B"])
    return F,U,D,R,L,B # return tuple

init_F, init_U, init_D, init_R, init_L, init_B=set_current_state({"F":"....W..W.", "U
   ":"...W...W.", "D":".......W.", "R":"..W...W..", "L":"......W..", "B":"..W
   ......"})

for STEPS in range(1, 20):
        print "trying %d steps" % STEPS

        s=Solver()
        state=[[[Bool('state%d_%d_%d' % (n, side, i)) for i in range(FACELETS)] for
           side in range(FACES)] for n in range(STEPS+1)]

        op=[Int('op%d' % n) for n in range(STEPS+1)]

        # initial state
        for i in range(FACELETS):
                s.add(state[0][0][i]==init_F[i])
                s.add(state[0][1][i]==init_U[i])
                s.add(state[0][2][i]==init_D[i])
                s.add(state[0][3][i]==init_R[i])
                s.add(state[0][4][i]==init_L[i])
                s.add(state[0][5][i]==init_B[i])

        # "must be" state for one (front/white) face
        for j in range(FACELETS):
                s.add(state[STEPS][0][j]==True)

        for n in range(STEPS):
                for side in range(FACES):
                        for j in range(FACELETS):
                                s.add(state[n+1][side][j]==rotate(op[n], state[n],
                                   side, j))

        if s.check()==sat:
                print "sat"
                m=s.model()
                for n in range(STEPS):
                        print move_names[int(str(m[op[n]]))]
```

```
                    exit(0)
```

(The full source code: https://smt.st/current_tree/puzzles/rubik3/one_face_SMT/rubik3_z3.py)

```
trying 1 steps
trying 2 steps
trying 3 steps
trying 4 steps
trying 5 steps
trying 6 steps
trying 7 steps
trying 8 steps
sat
LCW
UCCW
BCW
LCW
RCCW
BCCW
UCW
LCCW
```

Now the fun statistics. Using random walk I collected 928 states and then I tried to solve one (white/front) face for each state.

```
      1 turns= 4
      5 turns= 5
     57 turns= 6
    307 turns= 7
    501 turns= 8
     56 turns= 9
      1 turns= 10
```

It seems that majority of all states can be solved for 7-8 half turns (half-turn is one of 18 turns we used here). But there is at least one state which must be solved with 10 half turns. Maybe 10 is a ”god’s number” for one face, like 20 for all 6 faces?

8.8 Numberlink

8.8.1 Numberlink (AKA Flow Free) puzzle (Z3Py)

You probably saw Flow Free puzzle:

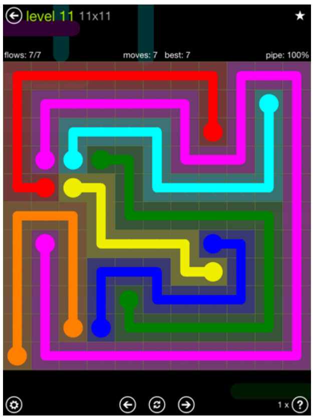


Figure 8.9

I’ll stick to Numberlink version of the puzzle. This is the example puzzle from Wikipedia:

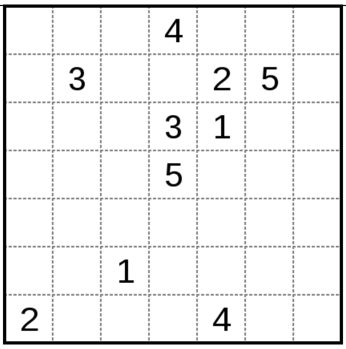


Figure 8.10

This is solved:

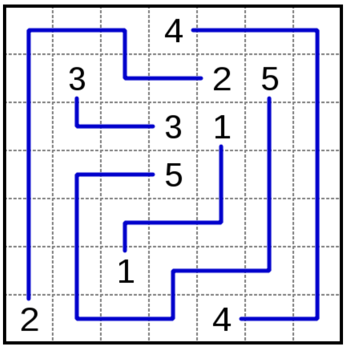


Figure 8.11

See also: https://en.wikipedia.org/wiki/Numberlink, https://en.wikipedia.org/wiki/Flow_Free.
The code:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

from z3 import *

puzzle=["   4   ",
" 3  25 ",
"   31  ",
"   5   ",
"       ",
"  1    ",
"2   4  "]

width=len(puzzle[0])
height=len(puzzle)

# number for each cell:
cells=[[Int('cell_r%d_c%d' % (r,c)) for c in range(width)] for r in range(height)]

# connections between cells. L means the cell has connection with cell at left, etc:
L=[[Bool('L_r%d_c%d' % (r,c)) for c in range(width)] for r in range(height)]
R=[[Bool('R_r%d_c%d' % (r,c)) for c in range(width)] for r in range(height)]
U=[[Bool('U_r%d_c%d' % (r,c)) for c in range(width)] for r in range(height)]
D=[[Bool('D_r%d_c%d' % (r,c)) for c in range(width)] for r in range(height)]

s=Solver()

# U for a cell must be equal to D of the cell above, etc:
for r in range(height):
    for c in range(width):
        if r!=0:
            s.add(U[r][c]==D[r-1][c])
        if r!=height-1:
            s.add(D[r][c]==U[r+1][c])
        if c!=0:
            s.add(L[r][c]==R[r][c-1])
        if c!=width-1:
            s.add(R[r][c]==L[r][c+1])

# yes, I know, we have 4 bools for each cell at this point, and we can half this number,
# but anyway, for the sake of simplicity, this could be better.

for r in range(height):
    for c in range(width):
        t=puzzle[r][c]
        if t==' ':
            # puzzle has space, so degree=2, IOW, this cell must have 2 connections, no more,
                no less.
            # enumerate all possible L/R/U/D booleans. two of them must be True, others are
                False.
            t=[]
            t.append(And(L[r][c], R[r][c], Not(U[r][c]), Not(D[r][c])))
            t.append(And(L[r][c], Not(R[r][c]), U[r][c], Not(D[r][c])))
            t.append(And(L[r][c], Not(R[r][c]), Not(U[r][c]), D[r][c]))
            t.append(And(Not(L[r][c]), R[r][c], U[r][c], Not(D[r][c])))
            t.append(And(Not(L[r][c]), R[r][c], Not(U[r][c]), D[r][c]))
            t.append(And(Not(L[r][c]), Not(R[r][c]), U[r][c], D[r][c]))
            s.add(Or(*t))
        else:
            # puzzle has number, add it to cells[][] as a constraint:
            s.add(cells[r][c]==int(t))
            # cell has degree=1, IOW, this cell must have 1 connection, no more, no less
            # enumerate all possible ways:
            t=[]
            t.append(And(L[r][c], Not(R[r][c]), Not(U[r][c]), Not(D[r][c])))
            t.append(And(Not(L[r][c]), R[r][c], Not(U[r][c]), Not(D[r][c])))
            t.append(And(Not(L[r][c]), Not(R[r][c]), U[r][c], Not(D[r][c])))
            t.append(And(Not(L[r][c]), Not(R[r][c]), Not(U[r][c]), D[r][c]))
            s.add(Or(*t))

        # if L[][]==True, cell's number must be equal to the number of cell at left, etc:
        if c!=0:
            s.add(If(L[r][c], cells[r][c]==cells[r][c-1], True))
        if c!=width-1:
            s.add(If(R[r][c], cells[r][c]==cells[r][c+1], True))
        if r!=0:
            s.add(If(U[r][c], cells[r][c]==cells[r-1][c], True))
        if r!=height-1:
            s.add(If(D[r][c], cells[r][c]==cells[r+1][c], True))

# L/R/U/D at borders sometimes must be always False:
for r in range(height):
    s.add(L[r][0]==False)
    s.add(R[r][width-1]==False)

for c in range(width):
    s.add(U[0][c]==False)
    s.add(D[height-1][c]==False)

# print solution:

print (s.check())
m=s.model()

print ("")

for r in range(height):
    for c in range(width):
        print (m[cells[r][c]],end=' ')
    print ("")

print ("")

for r in range(height):
    for c in range(width):
        t=""
        t=t+("L" if str(m[L[r][c]])=="True" else " ")
        t=t+("R" if str(m[R[r][c]])=="True" else " ")
        t=t+("U" if str(m[U[r][c]])=="True" else " ")
        t=t+("D" if str(m[D[r][c]])=="True" else " ")
        print (t+"|",end=' ')
    print ("")

print ("")

for r in range(height):
    row=""
    for c in range(width):
        t=puzzle[r][c]
        if t==' ':
            tl=(True if str(m[L[r][c]])=="True" else False)
            tr=(True if str(m[R[r][c]])=="True" else False)
            tu=(True if str(m[U[r][c]])=="True" else False)
            td=(True if str(m[D[r][c]])=="True" else False)

            if tu and td:
                row=row+"┃"
            if tr and td:
                row=row+"┏"
            if tr and tu:
                row=row+"┗"
            if tl and td:
                row=row+"┓"
            if tl and tu:
                row=row+"┛"
            if tl and tr:
                row=row+"━"
        else:
            row=row+t
    print (row)
```

The solution:

```
sat

2 2 2 4 4 4 4
2 3 2 2 2 5 4
2 3 3 3 1 5 4
2 5 5 5 1 5 4
2 5 1 1 1 5 4
2 5 1 5 5 5 4
2 5 5 5 4 4 4

 R D| LR  | L  D|  R  | LR  | LR  | L  D|
  UD|    D|  RU | LR  | L   |    D|   UD|
  UD|  RU | LR  | L   |    D|   UD|   UD|
  UD|  R D| LR  | L   |   UD|   UD|   UD|
  UD|   UD|  R D| LR  | L U |   UD|   UD|
  UD|   UD|   U |  R D| LR  | L U |   UD|
  U |  RU | LR  | L U |  R  | LR  | L U |
```

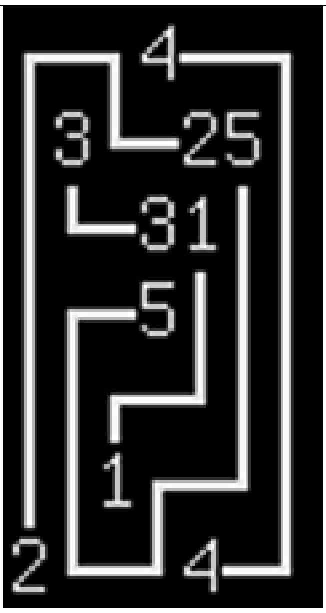


Figure 8.12: The solution

8.8.2 Numberlink (AKA Flow Free) puzzle as a MaxSAT problem + toy PCB router

Let's revisit my solution for Numberlink (AKA Flow Free) puzzle written for Z3Py.

What if holes in the puzzle exist? Can we make all paths as short as possible?

I've rewritten the puzzle solver using my own SAT library and now I use Open-WBO MaxSAT solver, see the source code, which is almost the same: `https://smt.st/current_tree/puzzles/numberlink/MaxSAT/numberlink_WBO.py`.

But now we "maximize" number of empty cells:

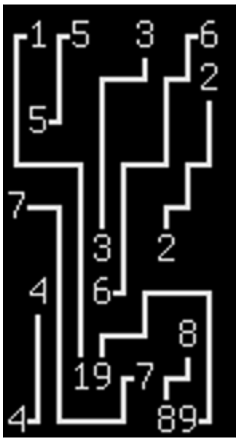


Figure 8.13

This is a solution with shortest possible paths. Others are possible, but their sum wouldn't be shorter. This is like toy PCB routing.

What if we comment the `s.fix_soft_always_true(cell_is_empty[r][c], 1)` line and set `maxsat=True`?

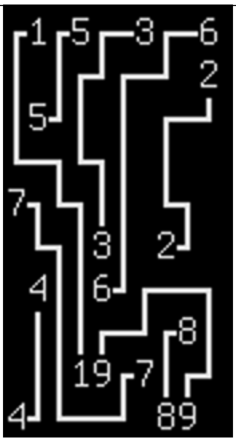


Figure 8.14

Lines 2 and 3 “roaming” chaotically, but the solution is correct, under given constraints.
The files: https://smt.st/current_tree/puzzles/numberlink/MaxSAT.

8.9 1959 AHSME Problems, Problem 19

From http://artofproblemsolving.com/wiki/index.php?title=1959_AHSME_Problems:

> With the use of three different weights, namely 1 lb., 3 lb., and 9 lb., how many objects of different weights can be weighed, if the objects is to be weighed and the given weights may be placed in either pan of the scale? (A) 15 (B) 13 (C) 11 (D) 9 (E) 7

This is fun!

```
from z3 import *

# 0 - weight absent, 1 - on left pan, 2 - on right pan:
w1, w3, w9, obj = Ints('w1 w3 w9 obj')

obj_w = Int('obj_w')

s=Solver()

s.add(And(w1>=0, w1<=2))
s.add(And(w3>=0, w3<=2))
s.add(And(w9>=0, w9<=2))

# object is always on left or right pan:
s.add(And(obj>=1, obj<=2))

# object must weight something:
s.add(obj_w>0)

left, right = Ints('left right')

# left pan is a sum of weights/object, if they are present on pan:
s.add(left  == If(w1==1, 1, 0) + If(w3==1, 3, 0) + If(w9==1, 9, 0) + If(obj==1, obj_w
    , 0))
# same for right pan:
s.add(right == If(w1==2, 1, 0) + If(w3==2, 3, 0) + If(w9==2, 9, 0) + If(obj==2, obj_w
    , 0))

# both pans must weight something:
```

```
s.add(left>0)
s.add(right>0)

# pans must have equal weights:
s.add(left==right)

# get all results:
results=[]
while True:
    if s.check() == sat:
        m = s.model()
        #print m
        print "left: ",
        print ("w1" if m[w1].as_long()==1 else "  "),
        print ("w3" if m[w3].as_long()==1 else "  "),
        print ("w9" if m[w9].as_long()==1 else "  "),
        print (("obj_w=%2d" % m[obj_w].as_long()) if m[obj].as_long()==1 else "
             "),

        print "    | right: ",
        print ("w1" if m[w1].as_long()==2 else "  "),
        print ("w3" if m[w3].as_long()==2 else "  "),
        print ("w9" if m[w9].as_long()==2 else "  "),
        print (("obj_w=%2d" % m[obj_w].as_long()) if m[obj].as_long()==2 else "
             "),
        print ""

        results.append(m)
        block = []
        for d in m:
            # skip internal variables, do not add them to blocking constraint:
            if str(d).startswith ("z3name"):
                continue
            c=d()
            block.append(c != m[d])
        s.add(Or(block))
    else:
        print "total results", len(results)
        break
```

Listing 8.7: Output

```
left:     w3                 | right:  w1       obj_w= 2
left:     w3    obj_w= 7     | right:  w1    w9
left:     w3 w9              | right:  w1       obj_w=11
left:     w3    obj_w= 6     | right:        w9
left:  w1 w3    obj_w= 5     | right:        w9
left:     w3 w9              | right:           obj_w=12
left:  w1 w3 w9              | right:           obj_w=13
left:  w1 w3                 | right:           obj_w= 4
left:     w3                 | right:           obj_w= 3
left:           obj_w= 4     | right:  w1 w3
left:           obj_w=13     | right:  w1 w3 w9
left:           obj_w= 3     | right:     w3
left:  w1       obj_w= 2     | right:     w3
left:  w1       obj_w=11     | right:     w3 w9
left:           obj_w=12     | right:     w3 w9
left:  w1                    | right:           obj_w= 1
left:        w9              | right:  w1       obj_w= 8
left:        w9              | right:           obj_w= 9
left:  w1    w9              | right:           obj_w=10
```

```
left:  w1    w9              | right:      w3    obj_w= 7
left:        w9              | right:  w1 w3    obj_w= 5
left:        w9              | right:      w3    obj_w= 6
left:           obj_w= 9     | right:         w9
left:           obj_w=10     | right:  w1     w9
left:           obj_w= 1     | right:  w1
left:  w1       obj_w= 8     | right:         w9
total results 26
```

There are 13 distinct `obj_w` values. So this is the answer.

By the way, an interesting problem I've found in the Simon Singh — Fermat's Last Theorem (1997) book:

> The *Arithmetica* which inspired Fermat was a Latin translation made by Claude Gaspar Bachet de Méziriac, reputedly the most learned man in all of France. As well as being a brilliant linguist, poet and classics scholar, Bachet had a passion for mathematical puzzles. His first publication was a compilation of puzzles entitled *Problemes plaisans et délectables qui se font par les nombres*, which included river-crossing problems, a liquid-pouring problem and several think-of-a-number tricks. One of the questions posed was a problem about weights:
>
> > What is the least number of weights that can be used on a set of scales to weigh any whole number of kilograms from 1 to 40
>
> Bachet had a cunning solution which shows that it is possible to achieve this task with only four weights...
>
> ...
>
> In order to weigh any whole number of kilograms from 1 to 40 most people will suggest that six weights are required: 1, 2, 4, 8, 16, 32 kg. In this way, all the weights can easily be achieved by placing the following combinations in one pan:
>
> 1 kg = 1,
> 2 kg = 2,
> 3 kg = 2 + 1,
> 4 kg = 4,
> 5 kg = 4 + 1,
> ...
> 40 kg = 32 + 8.
>
> However, by placing weights in both pans, such that weights are also allowed to sit alongside the object being weighed, Bachet could complete the task with only four weights: 1, 3, 9, 27 kg. A weight placed in the same pan as the object being weighed effectively assumes a negative value. Thus, the weights can be achieved as follows:
>
> 1 kg = 1,
> 2 kg = 3 - 1,
> 3 kg = 3,
> 4 kg = 3 + 1,
> 5 kg = 9 - 3 - 1,
> ...
> 40 kg = 27 + 9 + 3 + 1.

8.10 Alphametics

According to Donald Knuth, the term “Alphametics” was coined by J. A. H. Hunter. This is a puzzle: what decimal digits in 0..9 range must be assigned to each letter, so the following equation will be true?

```
  SEND
+ MORE
 -----
 MONEY
```

So is easy for Z3:

```
#!/usr/bin/env python3
from z3 import *

# SEND+MORE=MONEY

D, E, M, N, O, R, S, Y = Ints('D, E, M, N, O, R, S, Y')

s=Solver()

s.add(Distinct(D, E, M, N, O, R, S, Y))
s.add(And(D>=0, D<=9))
s.add(And(E>=0, E<=9))
s.add(And(M>=0, M<=9))
s.add(And(N>=0, N<=9))
s.add(And(O>=0, O<=9))
s.add(And(R>=0, R<=9))
s.add(And(S>=0, S<=9))
s.add(And(Y>=0, Y<=9))

s.add(1000*S+100*E+10*N+D + 1000*M+100*O+10*R+E == 10000*M+1000*O+100*N+10*E+Y)

print (s.check())
print (s.model())
```

Output:

```
sat
[E, = 5,
 S, = 9,
 M, = 1,
 N, = 6,
 D, = 7,
 R, = 8,
 O, = 0,
 Y = 2]
```

Another one, also from TAOCP volume IV (http://www-cs-faculty.stanford.edu/~uno/fasc2b.ps.gz):

```
#!/usr/bin/env python3
from z3 import *

# VIOLIN+VIOLIN+VIOLA = TRIO+SONATA

A, I, L, N, O, R, S, T, V = Ints('A, I, L, N, O, R, S, T, V')

s=Solver()

s.add(Distinct(A, I, L, N, O, R, S, T, V))
s.add(And(A>=0, A<=9))
s.add(And(I>=0, I<=9))
s.add(And(L>=0, L<=9))
s.add(And(N>=0, N<=9))
```

```
s.add(And(O>=0, O<=9))
s.add(And(R>=0, R<=9))
s.add(And(S>=0, S<=9))
s.add(And(T>=0, T<=9))
s.add(And(V>=0, V<=9))

VIOLIN, VIOLA, SONATA, TRIO = Ints('VIOLIN, VIOLA, SONATA, TRIO')

s.add(VIOLIN==100000*V+10000*I+1000*O+100*L+10*I+N)
s.add(VIOLA==10000*V+1000*I+100*O+10*L+A)
s.add(SONATA==100000*S+10000*O+1000*N+100*A+10*T+A)
s.add(TRIO==1000*T+100*R+10*I+O)

s.add(VIOLIN+VIOLIN+VIOLA == TRIO+SONATA)

print (s.check())
print (s.model())
```

```
sat
[L, = 6,
 S, = 7,
 N, = 2,
 T, = 1,
 I, = 5,
 V = 3,
 A, = 8,
 R, = 9,
 O, = 4,
 TRIO = 1954,
 SONATA, = 742818,
 VIOLA, = 35468,
 VIOLIN, = 354652]
```

This puzzle I've found in pySMT examples[9]:

```
#!/usr/bin/env python3
from z3 import *

# H+E+L+L+O = W+O+R+L+D = 25

H, E, L, O, W, R, D = Ints ('H E L O W R D')

s=Solver()

s.add(Distinct(H, E, L, O, W, R, D))
s.add(And(H>=1, H<=9))
s.add(And(E>=1, E<=9))
s.add(And(L>=1, L<=9))
s.add(And(O>=1, O<=9))
s.add(And(W>=1, W<=9))
s.add(And(R>=1, R<=9))
s.add(And(D>=1, D<=9))

s.add(H+E+L+L+O == 25)
s.add(W+O+R+L+D == 25)

print (s.check())
print (s.model())
```

[9] http://pysmt.readthedocs.io/en/latest/getting_started.html

```
sat
[D = 5, R = 4, O = 3, E = 8, L = 6, W = 7, H = 2]
```

This is an exercise Q209 from the [Companion to the Papers of Donald Knuth][10].

```
 KNIFE
  FORK
 SPOON
  SOUP
------
SUPPER
```

I've added a helper function (list_to_expr()) to make things simpler:

```
#!/usr/bin/env python3
from z3 import *

# KNIFE+FORK+SPOON+SOUP = SUPPER

E, F, I, K, N, O, P, R, S, U = Ints('E F I K N O P R S U')

s=Solver()

s.add(Distinct(E, F, I, K, N, O, P, R, S, U))
s.add(And(E>=0, E<=9))
s.add(And(F>=0, F<=9))
s.add(And(I>=0, I<=9))
s.add(And(K>=0, K<=9))
s.add(And(N>=0, N<=9))
s.add(And(O>=0, O<=9))
s.add(And(P>=0, P<=9))
s.add(And(R>=0, R<=9))
s.add(And(S>=0, S<=9))
s.add(And(U>=0, U<=9))

#s.add(S!=0)

KNIFE, FORK, SPOON, SOUP, SUPPER = Ints('KNIFE FORK SPOON SOUP SUPPER')

# construct expression in form like:
# 10000000*L+1000000*U+100000*N+10000*C+1000*H+100*E+10*O+N
def list_to_expr(lst):
    coeff=1
    _sum=0
    for var in lst[::-1]:
        _sum=_sum+var*coeff
        coeff=coeff*10
    return _sum

s.add(KNIFE==list_to_expr([K,N,I,F,E]))
s.add(FORK==list_to_expr([F,O,R,K]))
s.add(SPOON==list_to_expr([S,P,O,O,N]))
s.add(SOUP==list_to_expr([S,O,U,P]))
s.add(SUPPER==list_to_expr([S,U,P,P,E,R]))

s.add(KNIFE+FORK+SPOON+SOUP == SUPPER)

print (s.check())
print (s.model())
```

[10] http://www-cs-faculty.stanford.edu/~knuth/cp.html

```
sat
[K = 7,
 N = 4,
 R = 9,
 I = 1,
 E = 6,
 S = 0,
 O = 3,
 F = 5,
 U = 8,
 P = 2,
 SUPPER = 82269,
 SOUP = 382,
 SPOON = 2334,
 FORK = 5397,
 KNIFE = 74156]
```

S is zero, so SUPPER value is starting with leading (removed) zero. Let's say, we don't like it. Add this to resolve it:

```
s.add(S!=0)
```

```
sat
[K = 8,
 N = 4,
 R = 3,
 I = 7,
 E = 6,
 S = 1,
 O = 9,
 F = 2,
 U = 0,
 P = 5,
 SUPPER = 105563,
 SOUP = 1905,
 SPOON = 15994,
 FORK = 2938,
 KNIFE = 84726]
```

... but. How hard it is just to bruteforce? Not so hard: $10! = 3628800$ possible ways to assign 10 digits to 10 characters.

See also: Cryptarithmetic Puzzle Solver.

8.10.1 The files

https://smt.st/current_tree/puzzles/alphametics.

8.11 2015 AIME II Problems/Problem 12

At http://artofproblemsolving.com/wiki/index.php?title=2015_AIME_II_Problems/Problem_12:

> There are $2^{10} = 1024$ possible 10-letter strings in which each letter is either an A or a B. Find the number of such strings that do not have more than 3 adjacent letters that are identical.

We just find all 10-bit numbers, which don't have 4-bit runs of zeros or ones:

```
from z3 import *
```

```
a = BitVec('a', 10)

s=Solver()

for i in range(10-4+1):
    s.add(((a>>i)&15)!=0)
    s.add(((a>>i)&15)!=15)

results=[]
while True:
    if s.check() == sat:
        m = s.model()
        print "0x%x" % m[a].as_long()

        results.append(m)
        block = []
        for d in m:
            c=d()
            block.append(c != m[d])
        s.add(Or(block))
    else:
        print "total results", len(results)
        break
```

It's 548.

8.12 Fred puzzle

Found this:

```
Three fellows accused of stealing CDs make the following statements:

(1) Ed: “Fred did it, and Ted is innocent”.
(2) Fred: “If Ed is guilty, then so is Ted”.
(3) Ted: “’Im innocent, but at least one of the others is guilty”.

If the innocent told the truth and the guilty lied, who is guilty? (Remember that
    false statements imply anything).

I think Ed and Ted are innocent and Fred is guilty. Is it in contradiction with
    statement 2.

What do you say?
```

(https://math.stackexchange.com/questions/15199/implication-of-three-statements)
And how to convert this into logic statements:

```
Let us write the following propositions:

Fg means Fred is guilty, and Fi means Fred is innocent, Tg and Ti for Ted and Eg and
    Ei for Ed.

1. Ed says: Fg ∧ Ti
2. Fred says: Eg → Tg
3. Ted says: Ti ∧ (Fg ∨ Eg)

We know that the guilty is lying and the innocent tells the truth.

...
```

This is how I can implement it using Z3Py:

```
#!/usr/bin/env python3

from z3 import *

fg, fi, tg, ti, eg, ei = Bools('fg fi tg ti eg ei')

s=Solver()

s.add(fg!=fi)
s.add(tg!=ti)
s.add(eg!=ei)

s.add(ei==And(fg, ti))

s.add(fi==Implies(eg, tg))
#s.add(fi==Or(Not(eg), tg)) # Or(-x, y) is the same as Implies

s.add(ti==And(ti, Or(fg, eg)))

print (s.check())
print (s.model())
```

The result:

```
sat
[fg = False,
 ti = False,
 tg = True,
 eg = True,
 ei = False,
 fi = True]
```

(Fred is innocent, others are guilty.)

(`Implies` can be replaced with `Or(Not(x), y)`.)

Now in SMT-LIB v2 form:

```
; tested with Z3 and MK85:

(set-logic QF_BV)
(set-info :smt-lib-version 2.0)

(declare-fun fg () Bool)
(declare-fun fi () Bool)
(declare-fun tg () Bool)
(declare-fun ti () Bool)
(declare-fun eg () Bool)
(declare-fun ei () Bool)

(assert (not (= fg fi)))
(assert (not (= tg ti)))
(assert (not (= eg ei)))

(assert (= ei (and fg ti)))

; Or(-x, y) is the same as Implies
(assert (= fi (or (not eg) tg)))

(assert (= ti (and ti (or fg eg))))

(check-sat)
(get-model)
```

Again, it's small enough to be solved by MK85:

```
$ MK85 --dump-internal-variables fred.smt2
sat
(model
        (define-fun always_false () Bool false) ; var_no=1
        (define-fun always_true () Bool true) ; var_no=2
        (define-fun fg () Bool false) ; var_no=3
        (define-fun fi () Bool true) ; var_no=4
        (define-fun tg () Bool true) ; var_no=5
        (define-fun ti () Bool false) ; var_no=6
        (define-fun eg () Bool true) ; var_no=7
        (define-fun ei () Bool false) ; var_no=8
        (define-fun internal!1 () Bool true) ; var_no=9
        (define-fun internal!2 () Bool false) ; var_no=10
        (define-fun internal!3 () Bool true) ; var_no=11
        (define-fun internal!4 () Bool true) ; var_no=12
        (define-fun internal!5 () Bool false) ; var_no=13
        (define-fun internal!6 () Bool true) ; var_no=14
        (define-fun internal!7 () Bool true) ; var_no=15
        (define-fun internal!8 () Bool false) ; var_no=16
        (define-fun internal!9 () Bool true) ; var_no=17
        (define-fun internal!10 () Bool false) ; var_no=18
        (define-fun internal!11 () Bool false) ; var_no=19
        (define-fun internal!12 () Bool true) ; var_no=20
        (define-fun internal!13 () Bool false) ; var_no=21
        (define-fun internal!14 () Bool true) ; var_no=22
        (define-fun internal!15 () Bool false) ; var_no=23
        (define-fun internal!16 () Bool true) ; var_no=24
        (define-fun internal!17 () Bool true) ; var_no=25
        (define-fun internal!18 () Bool false) ; var_no=26
        (define-fun internal!19 () Bool false) ; var_no=27
        (define-fun internal!20 () Bool true) ; var_no=28
)
```

What is in the CNF file generated by MK85?

```
p cnf 28 64
-1 0
2 0
c generate_EQ id1=fg, id2=fi, var1=3, var2=4
c generate_XOR id1=fg id2=fi var1=3 var2=4 out id=internal!1 out var=9
-3 -4 -9 0
3 4 -9 0
3 -4 9 0
-3 4 9 0
c generate_NOT id=internal!1 var=9, out id=internal!2 out var=10
-10 -9 0
10 9 0
c generate_NOT id=internal!2 var=10, out id=internal!3 out var=11
-11 -10 0
11 10 0
c create_assert() id=internal!3 var=11
11 0
c generate_EQ id1=tg, id2=ti, var1=5, var2=6
c generate_XOR id1=tg id2=ti var1=5 var2=6 out id=internal!4 out var=12
-5 -6 -12 0
5 6 -12 0
5 -6 12 0
-5 6 12 0
c generate_NOT id=internal!4 var=12, out id=internal!5 out var=13
-13 -12 0
```

```
13 12 0
c generate_NOT id=internal!5 var=13, out id=internal!6 out var=14
-14 -13 0
14 13 0
c create_assert() id=internal!6 var=14
14 0
c generate_EQ id1=eg, id2=ei, var1=7, var2=8
c generate_XOR id1=eg id2=ei var1=7 var2=8 out id=internal!7 out var=15
-7 -8 -15 0
7 8 -15 0
7 -8 15 0
-7 8 15 0
c generate_NOT id=internal!7 var=15, out id=internal!8 out var=16
-16 -15 0
16 15 0
c generate_NOT id=internal!8 var=16, out id=internal!9 out var=17
-17 -16 0
17 16 0
c create_assert() id=internal!9 var=17
17 0
c generate_AND id1=fg id2=ti var1=3 var2=6 out id=internal!10 out var=18
-3 -6 18 0
3 -18 0
6 -18 0
c generate_EQ id1=ei, id2=internal!10, var1=8, var2=18
c generate_XOR id1=ei id2=internal!10 var1=8 var2=18 out id=internal!11 out var=19
-8 -18 -19 0
8 18 -19 0
8 -18 19 0
-8 18 19 0
c generate_NOT id=internal!11 var=19, out id=internal!12 out var=20
-20 -19 0
20 19 0
c create_assert() id=internal!12 var=20
20 0
c generate_NOT id=eg var=7, out id=internal!13 out var=21
-21 -7 0
21 7 0
c generate_OR id1=internal!13 id2=tg var1=21 var2=5 out id=internal!14 out var=22
21 5 -22 0
-21 22 0
-5 22 0
c generate_EQ id1=fi, id2=internal!14, var1=4, var2=22
c generate_XOR id1=fi id2=internal!14 var1=4 var2=22 out id=internal!15 out var=23
-4 -22 -23 0
4 22 -23 0
4 -22 23 0
-4 22 23 0
c generate_NOT id=internal!15 var=23, out id=internal!16 out var=24
-24 -23 0
24 23 0
c create_assert() id=internal!16 var=24
24 0
c generate_OR id1=fg id2=eg var1=3 var2=7 out id=internal!17 out var=25
3 7 -25 0
-3 25 0
-7 25 0
c generate_AND id1=ti id2=internal!17 var1=6 var2=25 out id=internal!18 out var=26
-6 -25 26 0
6 -26 0
25 -26 0
```

```
c generate_EQ id1=ti, id2=internal!18, var1=6, var2=26
c generate_XOR id1=ti id2=internal!18 var1=6 var2=26 out id=internal!19 out var=27
-6 -26 -27 0
6 26 -27 0
6 -26 27 0
-6 26 27 0
c generate_NOT id=internal!19 var=27, out id=internal!20 out var=28
-28 -27 0
28 27 0
c create_assert() id=internal!20 var=28
28 0
```

Let's filter out comments:

```
c generate_EQ id1=fg, id2=fi, var1=3, var2=4
c generate_XOR id1=fg id2=fi var1=3 var2=4 out id=internal!1 out var=9
c generate_NOT id=internal!1 var=9, out id=internal!2 out var=10
c generate_NOT id=internal!2 var=10, out id=internal!3 out var=11
c create_assert() id=internal!3 var=11
c generate_EQ id1=tg, id2=ti, var1=5, var2=6
c generate_XOR id1=tg id2=ti var1=5 var2=6 out id=internal!4 out var=12
c generate_NOT id=internal!4 var=12, out id=internal!5 out var=13
c generate_NOT id=internal!5 var=13, out id=internal!6 out var=14
c create_assert() id=internal!6 var=14
c generate_EQ id1=eg, id2=ei, var1=7, var2=8
c generate_XOR id1=eg id2=ei var1=7 var2=8 out id=internal!7 out var=15
c generate_NOT id=internal!7 var=15, out id=internal!8 out var=16
c generate_NOT id=internal!8 var=16, out id=internal!9 out var=17
c create_assert() id=internal!9 var=17
c generate_AND id1=fg id2=ti var1=3 var2=6 out id=internal!10 out var=18
c generate_EQ id1=ei, id2=internal!10, var1=8, var2=18
c generate_XOR id1=ei id2=internal!10 var1=8 var2=18 out id=internal!11 out var=19
c generate_NOT id=internal!11 var=19, out id=internal!12 out var=20
c create_assert() id=internal!12 var=20
c generate_NOT id=eg var=7, out id=internal!13 out var=21
c generate_OR id1=internal!13 id2=tg var1=21 var2=5 out id=internal!14 out var=22
c generate_EQ id1=fi, id2=internal!14, var1=4, var2=22
c generate_XOR id1=fi id2=internal!14 var1=4 var2=22 out id=internal!15 out var=23
c generate_NOT id=internal!15 var=23, out id=internal!16 out var=24
c create_assert() id=internal!16 var=24
c generate_OR id1=fg id2=eg var1=3 var2=7 out id=internal!17 out var=25
c generate_AND id1=ti id2=internal!17 var1=6 var2=25 out id=internal!18 out var=26
c generate_EQ id1=ti, id2=internal!18, var1=6, var2=26
c generate_XOR id1=ti id2=internal!18 var1=6 var2=26 out id=internal!19 out var=27
c generate_NOT id=internal!19 var=27, out id=internal!20 out var=28
c create_assert() id=internal!20 var=28
```

Again, this instance is small enough to be solved by small backtracking SAT-solver:

```
$ python SAT_backtrack.py tmp.cnf
SAT
-1 2 -3 4 5 -6 7 -8 9 -10 11 12 -13 14 15 -16 17 -18 -19 20 -21 22 -23 24 25 -26 -27
   28 0
UNSAT
solutions= 1
```

See also: solving *Fred puzzle* using regex matcher: 22.2.

8.13 Multiple choice logic puzzle

The source of this puzzle is probably Ross Honsberger's "More Mathematical Morsels (Dolciani Mathematical Expositions)" book:

```
A certain question has the following possible answers.

    a. All of the below
    b. None of the below
    c. All of the above
    d. One of the above
    e. None of the above
    f. None of the above

Which answer is correct?
```

Cited from: https://www.johndcook.com/blog/2015/07/06/multiple-choice/.

This problem can be easily represented as a system of boolean equations. Let's try to solve it using Z3. Each bool represents if the specific sentence is true.

```
#!/usr/bin/env python3

from z3 import *

a, b, c, d, e, f = Bools('a b c d e f')

s=Solver()

s.add(a==And(b,c,d,e,f))
s.add(b==And(Not(c),Not(d),Not(e),Not(f)))
s.add(c==And(a,b))
s.add(d==Or(And(a,Not(b),Not(c)), And(Not(a),b,Not(c)), And(Not(a),Not(b),c)))
s.add(e==And(Not(a),Not(b),Not(c),Not(d)))
s.add(f==And(Not(a),Not(b),Not(c),Not(d), Not(e)))

print (s.check())
print (s.model())
```

The answer:

```
sat
[f = False,
 b = False,
 a = False,
 c = False,
 d = False,
 e = True]
```

I can also rewrite this in SMT-LIB v2 form:

```
; tested with Z3 and MK85

(set-logic QF_BV)
(set-info :smt-lib-version 2.0)

(declare-fun a () Bool)
(declare-fun b () Bool)
(declare-fun c () Bool)
(declare-fun d () Bool)
(declare-fun e () Bool)
(declare-fun f () Bool)

(assert (= a (and b c d e f)))
(assert (= b (and (not c) (not d) (not e) (not f))))
(assert (= c (and a b)))
(assert (= d (or
        (and a (not b) (not c))
        (and (not a) b (not c))
        (and (not a) (not b) c)
        )))
(assert (= e (and (not a) (not b) (not c) (not d))))
(assert (= f (and (not a) (not b) (not c) (not d) (not e))))

(check-sat)
(get-model)
```

The problem is easy enough to be solved using MK85:

```
sat
(model
        (define-fun a () Bool false)
        (define-fun b () Bool false)
        (define-fun c () Bool false)
        (define-fun d () Bool false)
        (define-fun e () Bool true)
        (define-fun f () Bool false)
)
```

Now let's have fun and see how my toy SMT solver tackles this example. What internal variables it creates?

```
$ ./MK85 --dump-internal-variables mc.smt2

sat
(model
        (define-fun always_false () Bool false) ; var_no=1
        (define-fun always_true () Bool true) ; var_no=2
        (define-fun a () Bool false) ; var_no=3
        (define-fun b () Bool false) ; var_no=4
        (define-fun c () Bool false) ; var_no=5
        (define-fun d () Bool false) ; var_no=6
        (define-fun e () Bool true) ; var_no=7
        (define-fun f () Bool false) ; var_no=8
        (define-fun internal!1 () Bool false) ; var_no=9
        (define-fun internal!2 () Bool false) ; var_no=10
        (define-fun internal!3 () Bool false) ; var_no=11
        (define-fun internal!4 () Bool false) ; var_no=12
        (define-fun internal!5 () Bool false) ; var_no=13
        (define-fun internal!6 () Bool true) ; var_no=14
        (define-fun internal!7 () Bool true) ; var_no=15
        (define-fun internal!8 () Bool true) ; var_no=16
        (define-fun internal!9 () Bool true) ; var_no=17
        (define-fun internal!10 () Bool false) ; var_no=18
        (define-fun internal!11 () Bool false) ; var_no=19
        (define-fun internal!12 () Bool true) ; var_no=20
        (define-fun internal!13 () Bool false) ; var_no=21
        (define-fun internal!14 () Bool false) ; var_no=22
        (define-fun internal!15 () Bool true) ; var_no=23
        (define-fun internal!16 () Bool false) ; var_no=24
        (define-fun internal!17 () Bool false) ; var_no=25
        (define-fun internal!18 () Bool true) ; var_no=26
        (define-fun internal!19 () Bool true) ; var_no=27
        (define-fun internal!20 () Bool false) ; var_no=28
        (define-fun internal!21 () Bool true) ; var_no=29
        (define-fun internal!22 () Bool false) ; var_no=30
        (define-fun internal!23 () Bool true) ; var_no=31
        (define-fun internal!24 () Bool false) ; var_no=32
        (define-fun internal!25 () Bool true) ; var_no=33
        (define-fun internal!26 () Bool false) ; var_no=34
        (define-fun internal!27 () Bool false) ; var_no=35
        (define-fun internal!28 () Bool true) ; var_no=36
        (define-fun internal!29 () Bool true) ; var_no=37
        (define-fun internal!30 () Bool true) ; var_no=38
        (define-fun internal!31 () Bool false) ; var_no=39
        (define-fun internal!32 () Bool false) ; var_no=40
        (define-fun internal!33 () Bool false) ; var_no=41
        (define-fun internal!34 () Bool true) ; var_no=42
        (define-fun internal!35 () Bool true) ; var_no=43
        (define-fun internal!36 () Bool true) ; var_no=44
        (define-fun internal!37 () Bool true) ; var_no=45
        (define-fun internal!38 () Bool true) ; var_no=46
        (define-fun internal!39 () Bool true) ; var_no=47
        (define-fun internal!40 () Bool true) ; var_no=48
        (define-fun internal!41 () Bool true) ; var_no=49
        (define-fun internal!42 () Bool false) ; var_no=50
        (define-fun internal!43 () Bool true) ; var_no=51
        (define-fun internal!44 () Bool true) ; var_no=52
        (define-fun internal!45 () Bool true) ; var_no=53
        (define-fun internal!46 () Bool true) ; var_no=54
        (define-fun internal!47 () Bool true) ; var_no=55
        (define-fun internal!48 () Bool true) ; var_no=56
        (define-fun internal!49 () Bool true) ; var_no=57
        (define-fun internal!50 () Bool true) ; var_no=58
        (define-fun internal!51 () Bool false) ; var_no=59
        (define-fun internal!52 () Bool false) ; var_no=60
        (define-fun internal!53 () Bool false) ; var_no=61
        (define-fun internal!54 () Bool true) ; var_no=62
)
```

What is in resulting CNF file to be fed into the external picosat SAT-solver?

```
p cnf 62 151
-1 0
2 0
c generate_AND id1=b id2=c var1=4 var2=5 out id=internal!1 out var=9
-4 -5 9 0
4 -9 0
5 -9 0
c generate_AND id1=internal!1 id2=d var1=9 var2=6 out id=internal!2 out var=10
-9 -6 10 0
9 -10 0
6 -10 0
c generate_AND id1=internal!2 id2=e var1=10 var2=7 out id=internal!3 out var=11
-10 -7 11 0
10 -11 0
7 -11 0
c generate_AND id1=internal!3 id2=f var1=11 var2=8 out id=internal!4 out var=12
-11 -8 12 0
11 -12 0
8 -12 0
c generate_EQ id1=a, id2=internal!4, var1=3, var2=12
c generate_XOR id1=a id2=internal!4 var1=3 var2=12 out id=internal!5 out var=13
-3 -12 -13 0
3 12 -13 0
3 -12 13 0
-3 12 13 0
c generate_NOT id=internal!5 var=13, out id=internal!6 out var=14
-14 -13 0
14 13 0
c create_assert() id=internal!6 var=14
14 0
c generate_NOT id=c var=5, out id=internal!7 out var=15
-15 -5 0
15 5 0
```

```
c generate_NOT id=d var=6, out id=internal!8 out var=16
-16 -6 0
16 6 0
c generate_AND id1=internal!7 id2=internal!8 var1=15 var2=16 out id=internal!9 out
   var=17
-15 -16 17 0
15 -17 0
16 -17 0
c generate_NOT id=e var=7, out id=internal!10 out var=18
-18 -7 0
18 7 0
c generate_AND id1=internal!9 id2=internal!10 var1=17 var2=18 out id=internal!11 out
   var=19
-17 -18 19 0
17 -19 0
18 -19 0
c generate_NOT id=f var=8, out id=internal!12 out var=20
-20 -8 0
20 8 0
c generate_AND id1=internal!11 id2=internal!12 var1=19 var2=20 out id=internal!13 out
    var=21
-19 -20 21 0
19 -21 0
20 -21 0
c generate_EQ id1=b, id2=internal!13, var1=4, var2=21
c generate_XOR id1=b id2=internal!13 var1=4 var2=21 out id=internal!14 out var=22
-4 -21 -22 0
4 21 -22 0
4 -21 22 0
-4 21 22 0
c generate_NOT id=internal!14 var=22, out id=internal!15 out var=23
-23 -22 0
23 22 0
c create_assert() id=internal!15 var=23
23 0
c generate_AND id1=a id2=b var1=3 var2=4 out id=internal!16 out var=24
-3 -4 24 0
3 -24 0
4 -24 0
c generate_EQ id1=c, id2=internal!16, var1=5, var2=24
c generate_XOR id1=c id2=internal!16 var1=5 var2=24 out id=internal!17 out var=25
-5 -24 -25 0
5 24 -25 0
5 -24 25 0
-5 24 25 0
c generate_NOT id=internal!17 var=25, out id=internal!18 out var=26
-26 -25 0
26 25 0
c create_assert() id=internal!18 var=26
26 0
c generate_NOT id=b var=4, out id=internal!19 out var=27
-27 -4 0
27 4 0
c generate_AND id1=a id2=internal!19 var1=3 var2=27 out id=internal!20 out var=28
-3 -27 28 0
3 -28 0
27 -28 0
c generate_NOT id=c var=5, out id=internal!21 out var=29
-29 -5 0
29 5 0
c generate_AND id1=internal!20 id2=internal!21 var1=28 var2=29 out id=internal!22 out
```

```
    var=30
-28 -29 30 0
28 -30 0
29 -30 0
c generate_NOT id=a var=3, out id=internal!23 out var=31
-31 -3 0
31 3 0
c generate_AND id1=internal!23 id2=b var1=31 var2=4 out id=internal!24 out var=32
-31 -4 32 0
31 -32 0
4 -32 0
c generate_NOT id=c var=5, out id=internal!25 out var=33
-33 -5 0
33 5 0
c generate_AND id1=internal!24 id2=internal!25 var1=32 var2=33 out id=internal!26 out
    var=34
-32 -33 34 0
32 -34 0
33 -34 0
c generate_OR id1=internal!22 id2=internal!26 var1=30 var2=34 out id=internal!27 out
   var=35
30 34 -35 0
-30 35 0
-34 35 0
c generate_NOT id=a var=3, out id=internal!28 out var=36
-36 -3 0
36 3 0
c generate_NOT id=b var=4, out id=internal!29 out var=37
-37 -4 0
37 4 0
c generate_AND id1=internal!28 id2=internal!29 var1=36 var2=37 out id=internal!30 out
    var=38
-36 -37 38 0
36 -38 0
37 -38 0
c generate_AND id1=internal!30 id2=c var1=38 var2=5 out id=internal!31 out var=39
-38 -5 39 0
38 -39 0
5 -39 0
c generate_OR id1=internal!27 id2=internal!31 var1=35 var2=39 out id=internal!32 out
   var=40
35 39 -40 0
-35 40 0
-39 40 0
c generate_EQ id1=d, id2=internal!32, var1=6, var2=40
c generate_XOR id1=d id2=internal!32 var1=6 var2=40 out id=internal!33 out var=41
-6 -40 -41 0
6 40 -41 0
6 -40 41 0
-6 40 41 0
c generate_NOT id=internal!33 var=41, out id=internal!34 out var=42
-42 -41 0
42 41 0
c create_assert() id=internal!34 var=42
42 0
c generate_NOT id=a var=3, out id=internal!35 out var=43
-43 -3 0
43 3 0
c generate_NOT id=b var=4, out id=internal!36 out var=44
-44 -4 0
44 4 0
```

```
c generate_AND id1=internal!35 id2=internal!36 var1=43 var2=44 out id=internal!37 out
    var=45
-43 -44 45 0
43 -45 0
44 -45 0
c generate_NOT id=c var=5, out id=internal!38 out var=46
-46 -5 0
46 5 0
c generate_AND id1=internal!37 id2=internal!38 var1=45 var2=46 out id=internal!39 out
    var=47
-45 -46 47 0
45 -47 0
46 -47 0
c generate_NOT id=d var=6, out id=internal!40 out var=48
-48 -6 0
48 6 0
c generate_AND id1=internal!39 id2=internal!40 var1=47 var2=48 out id=internal!41 out
    var=49
-47 -48 49 0
47 -49 0
48 -49 0
c generate_EQ id1=e, id2=internal!41, var1=7, var2=49
c generate_XOR id1=e id2=internal!41 var1=7 var2=49 out id=internal!42 out var=50
-7 -49 -50 0
7 49 -50 0
7 -49 50 0
-7 49 50 0
c generate_NOT id=internal!42 var=50, out id=internal!43 out var=51
-51 -50 0
51 50 0
c create_assert() id=internal!43 var=51
51 0
c generate_NOT id=a var=3, out id=internal!44 out var=52
-52 -3 0
52 3 0
c generate_NOT id=b var=4, out id=internal!45 out var=53
-53 -4 0
53 4 0
c generate_AND id1=internal!44 id2=internal!45 var1=52 var2=53 out id=internal!46 out
    var=54
-52 -53 54 0
52 -54 0
53 -54 0
c generate_NOT id=c var=5, out id=internal!47 out var=55
-55 -5 0
55 5 0
c generate_AND id1=internal!46 id2=internal!47 var1=54 var2=55 out id=internal!48 out
    var=56
-54 -55 56 0
54 -56 0
55 -56 0
c generate_NOT id=d var=6, out id=internal!49 out var=57
-57 -6 0
57 6 0
c generate_AND id1=internal!48 id2=internal!49 var1=56 var2=57 out id=internal!50 out
    var=58
-56 -57 58 0
56 -58 0
57 -58 0
c generate_NOT id=e var=7, out id=internal!51 out var=59
-59 -7 0
```

```
59 7 0
c generate_AND id1=internal!50 id2=internal!51 var1=58 var2=59 out id=internal!52 out var=60
-58 -59 60 0
58 -60 0
59 -60 0
c generate_EQ id1=f, id2=internal!52, var1=8, var2=60
c generate_XOR id1=f id2=internal!52 var1=8 var2=60 out id=internal!53 out var=61
-8 -60 -61 0
8 60 -61 0
8 -60 61 0
-8 60 61 0
c generate_NOT id=internal!53 var=61, out id=internal!54 out var=62
-62 -61 0
62 61 0
c create_assert() id=internal!54 var=62
62 0
```

Here are comments (starting with "c " prefix), and my SMT-solver indicate, how each low-level logical gate is added, its inputs (variable IDs and numbers) and outputs.

Let's filter comments:

```
$ cat tmp.cnf | grep "^c "

c generate_AND id1=b id2=c var1=4 var2=5 out id=internal!1 out var=9
c generate_AND id1=internal!1 id2=d var1=9 var2=6 out id=internal!2 out var=10
c generate_AND id1=internal!2 id2=e var1=10 var2=7 out id=internal!3 out var=11
c generate_AND id1=internal!3 id2=f var1=11 var2=8 out id=internal!4 out var=12
c generate_EQ id1=a, id2=internal!4, var1=3, var2=12
c generate_XOR id1=a id2=internal!4 var1=3 var2=12 out id=internal!5 out var=13
c generate_NOT id=internal!5 var=13, out id=internal!6 out var=14
c create_assert() id=internal!6 var=14
c generate_NOT id=c var=5, out id=internal!7 out var=15
c generate_NOT id=d var=6, out id=internal!8 out var=16
c generate_AND id1=internal!7 id2=internal!8 var1=15 var2=16 out id=internal!9 out var=17
c generate_NOT id=e var=7, out id=internal!10 out var=18
c generate_AND id1=internal!9 id2=internal!10 var1=17 var2=18 out id=internal!11 out var=19
c generate_NOT id=f var=8, out id=internal!12 out var=20
c generate_AND id1=internal!11 id2=internal!12 var1=19 var2=20 out id=internal!13 out var=21
c generate_EQ id1=b, id2=internal!13, var1=4, var2=21
c generate_XOR id1=b id2=internal!13 var1=4 var2=21 out id=internal!14 out var=22
c generate_NOT id=internal!14 var=22, out id=internal!15 out var=23
c create_assert() id=internal!15 var=23
c generate_AND id1=a id2=b var1=3 var2=4 out id=internal!16 out var=24
c generate_EQ id1=c, id2=internal!16, var1=5, var2=24
c generate_XOR id1=c id2=internal!16 var1=5 var2=24 out id=internal!17 out var=25
c generate_NOT id=internal!17 var=25, out id=internal!18 out var=26
c create_assert() id=internal!18 var=26
c generate_NOT id=b var=4, out id=internal!19 out var=27
c generate_AND id1=a id2=internal!19 var1=3 var2=27 out id=internal!20 out var=28
c generate_NOT id=c var=5, out id=internal!21 out var=29
c generate_AND id1=internal!20 id2=internal!21 var1=28 var2=29 out id=internal!22 out var=30
c generate_NOT id=a var=3, out id=internal!23 out var=31
c generate_AND id1=internal!23 id2=b var1=31 var2=4 out id=internal!24 out var=32
c generate_NOT id=c var=5, out id=internal!25 out var=33
c generate_AND id1=internal!24 id2=internal!25 var1=32 var2=33 out id=internal!26 out var=34
c generate_OR id1=internal!22 id2=internal!26 var1=30 var2=34 out id=internal!27 out
```

```
    var=35
c generate_NOT id=a var=3, out id=internal!28 out var=36
c generate_NOT id=b var=4, out id=internal!29 out var=37
c generate_AND id1=internal!28 id2=internal!29 var1=36 var2=37 out id=internal!30 out
     var=38
c generate_AND id1=internal!30 id2=c var1=38 var2=5 out id=internal!31 out var=39
c generate_OR id1=internal!27 id2=internal!31 var1=35 var2=39 out id=internal!32 out
    var=40
c generate_EQ id1=d, id2=internal!32, var1=6, var2=40
c generate_XOR id1=d id2=internal!32 var1=6 var2=40 out id=internal!33 out var=41
c generate_NOT id=internal!33 var=41, out id=internal!34 out var=42
c create_assert() id=internal!34 var=42
c generate_NOT id=a var=3, out id=internal!35 out var=43
c generate_NOT id=b var=4, out id=internal!36 out var=44
c generate_AND id1=internal!35 id2=internal!36 var1=43 var2=44 out id=internal!37 out
     var=45
c generate_NOT id=c var=5, out id=internal!38 out var=46
c generate_AND id1=internal!37 id2=internal!38 var1=45 var2=46 out id=internal!39 out
     var=47
c generate_NOT id=d var=6, out id=internal!40 out var=48
c generate_AND id1=internal!39 id2=internal!40 var1=47 var2=48 out id=internal!41 out
     var=49
c generate_EQ id1=e, id2=internal!41, var1=7, var2=49
c generate_XOR id1=e id2=internal!41 var1=7 var2=49 out id=internal!42 out var=50
c generate_NOT id=internal!42 var=50, out id=internal!43 out var=51
c create_assert() id=internal!43 var=51
c generate_NOT id=a var=3, out id=internal!44 out var=52
c generate_NOT id=b var=4, out id=internal!45 out var=53
c generate_AND id1=internal!44 id2=internal!45 var1=52 var2=53 out id=internal!46 out
     var=54
c generate_NOT id=c var=5, out id=internal!47 out var=55
c generate_AND id1=internal!46 id2=internal!47 var1=54 var2=55 out id=internal!48 out
     var=56
c generate_NOT id=d var=6, out id=internal!49 out var=57
c generate_AND id1=internal!48 id2=internal!49 var1=56 var2=57 out id=internal!50 out
     var=58
c generate_NOT id=e var=7, out id=internal!51 out var=59
c generate_AND id1=internal!50 id2=internal!51 var1=58 var2=59 out id=internal!52 out
     var=60
c generate_EQ id1=f, id2=internal!52, var1=8, var2=60
c generate_XOR id1=f id2=internal!52 var1=8 var2=60 out id=internal!53 out var=61
c generate_NOT id=internal!53 var=61, out id=internal!54 out var=62
c create_assert() id=internal!54 var=62
```

Now you can juxtapose list of internal variables and comments in CNF file. For example, equality gate is generated as NOT(XOR(a,b)).

create_assert() function fixes a bool variable to be always True.

Other (internal) variables are added by SMT solver as “joints” to connect logic gates with each other.

Hence, my SMT solver constructing a digital circuit based on the input SMT file. Logic gates are then converted into CNF form using Tseitin transformations. The task of SAT solver is then to find such an assignment, for which CNF expression would be true. In other words, its task is to find such inputs/outputs for which this “virtual” digital circuit would work without contradictions.

The SAT instance is also small enough to be solved using my simplest backtracking SAT solver written:

```
$ ./SAT_backtrack.py tmp.cnf

SAT
-1 2 -3 -4 -5 -6 7 -8 -9 -10 -11 -12 -13 14 15 16 17 -18 -19 20 -21 -22 23 -24 -25 26
     27 -28 29 -30 31 -32 33 -34 -35 36 37 38 -39 -40 -41 42 43 44 45 46 47 48 49 -50
    51 52 53 54 55 56 57 58 -59 -60 -61 62 0
UNSAT
solutions= 1
```

You can juxtapose variables from solver's result and variable numbers from MK85 listing. Therefore, MK85 + my small SAT solver is standalone program under 3000 SLOC, which still can solve such (simple enough) system of boolean equations, without external aid like minisat/picosat.

Among comments at the John D. Cook's blog, there is also a solution by Aaron Meurer, using SymPy, which also has SAT-solver inside:

```
7 July 2015 at 01:34

Decided to run this through ’SymPys SAT solver.

In [1]: var‘(a b c d e ’f)
Out[1]: (a, b, c, d, e, f)

In [2]: facts = [
Equivalent(a, (b & c & d & e & f)),
Equivalent(b, (~c & ~d & ~e & ~f)),
Equivalent(c, a & b),
Equivalent(d, ((a & ~b & ~c) | (~a & b & ~c) | (~a & ~b & c))),
Equivalent(e, (~a & ~b & ~c & ~d)),
Equivalent(f, (~a & ~b & ~c & ~d & ~e)),
]

In [3]: list(satisfiable(And(*facts), all_models=True))
Out[3]: [{e: True, c: False, b: False, a: False, f: False, d: False}]

So it seems e is the only answer, assuming I got the facts correct. And it is
    important to use Equivalent (a bidirectional implication) rather than just Implies
    . If you only use -> (which I guess would mean that an answer not being chosen
    ’doesnt necessarily mean that it ’isnt true), then ‘’none, b, and f are also
    ‘“’solutions.

Also, if I replace the d fact with Equivalent(d, a | b | c), the result is the same.
    So it seems that the interpretation of ‘“’one both in terms of choice d and in
    terms of how many choices there are is irrelevant.

Thanks for the fun problem. I hope others took the time to solve this in their head
    before reading the comments.
```

8.14 Art of problem solving

http://artofproblemsolving.com/wiki/index.php?title=Mock_AIME_2_2006-2007_Problems/Problem_8:

> The positive integers $x_1, x_2, ..., x_7$ satisfy $x_6 = 144$ and $x_{n+3} = x_{n+2}(x_{n+1} + x_n)$ for $n = 1, 2, 3, 4$. Find the last three digits of x_7.

This is it:

```
from z3 import *

s=Solver()

x1, x2, x3, x4, x5, x6, x7=Ints('x1 x2 x3 x4 x5 x6 x7')

s.add(x1>=0)
```

```
s.add(x2>=0)
s.add(x3>=0)
s.add(x4>=0)
s.add(x5>=0)
s.add(x6>=0)
s.add(x7>=0)

s.add(x6==144)

s.add(x4==x3*(x2+x1))
s.add(x5==x4*(x3+x2))
s.add(x6==x5*(x4+x3))
s.add(x7==x6*(x5+x4))

# get all results:

results=[]
while True:
    if s.check() == sat:
        m = s.model()
        print m

        results.append(m)
        block = []
        for d in m:
            c=d()
            block.append(c != m[d])
        s.add(Or(block))
    else:
        print "total results", len(results)
        break
```

Two solutions possible, but in both x7 is ending by 456:

```
[x2 = 1,
 x3 = 1,
 x1 = 7,
 x4 = 8,
 x5 = 16,
 x7 = 3456,
 x6 = 144]
[x3 = 2,
 x2 = 1,
 x1 = 2,
 x6 = 144,
 x4 = 6,
 x5 = 18,
 x7 = 3456]
total results 2
```

8.15 2012 AIME I Problems/Problem 1

http://artofproblemsolving.com/wiki/index.php?title=2012_AIME_I_Problems/Problem_1

> Find the number of positive integers with three not necessarily distinct digits, abc, with $a \neq 0$ and $c \neq 0$ such that both abc and cba are multiples of 4.

```
from z3 import *

a, b, c = Ints('a b c')

s=Solver()

s.add(a>0)
s.add(b>=0)
s.add(c>0)

s.add(a<=9)
s.add(b<=9)
s.add(c<=9)

s.add((a*100 + b*10 + c) % 4 == 0)
s.add((c*100 + b*10 + a) % 4 == 0)

results=[]
while True:
    if s.check() == sat:
        m = s.model()
        print m

        results.append(m)
        block = []
        for d in m:
            c=d()
            block.append(c != m[d])
        s.add(Or(block))
    else:
        print "total results", len(results)
        break
```

Let's see:

```
[c = 4, b = 0, a = 4]
[b = 1, c = 2, a = 2]
[b = 6, c = 4, a = 4]
[b = 4, c = 4, a = 4]
[b = 2, c = 4, a = 4]
[b = 4, c = 4, a = 8]
[b = 8, c = 4, a = 4]
[b = 6, c = 4, a = 8]
[b = 8, c = 4, a = 8]
[b = 0, c = 4, a = 8]
[b = 2, c = 4, a = 8]
[b = 8, c = 8, a = 8]
[b = 9, c = 6, a = 6]
[b = 2, c = 8, a = 8]
[b = 2, c = 8, a = 4]
[b = 4, c = 8, a = 4]
[b = 4, c = 8, a = 8]
[b = 8, c = 8, a = 4]
[b = 6, c = 8, a = 4]
[b = 6, c = 8, a = 8]
[b = 0, c = 8, a = 4]
[b = 0, c = 8, a = 8]
[b = 7, c = 6, a = 6]
[b = 7, c = 2, a = 6]
[b = 7, c = 6, a = 2]
[b = 7, c = 2, a = 2]
```

```
[b = 9, c = 2, a = 6]
[b = 9, c = 2, a = 2]
[b = 9, c = 6, a = 2]
[b = 5, c = 6, a = 2]
[b = 1, c = 6, a = 2]
[b = 3, c = 6, a = 2]
[b = 5, c = 2, a = 2]
[b = 3, c = 2, a = 2]
[b = 5, c = 6, a = 6]
[b = 5, c = 2, a = 6]
[b = 3, c = 2, a = 6]
[b = 1, c = 2, a = 6]
[b = 1, c = 6, a = 6]
[b = 3, c = 6, a = 6]
total results 40
```

My toy-level SMT-solver MK85 can enumerate all solutions as well:

```
(set-logic QF_BV)
(set-info :smt-lib-version 2.0)

(declare-fun a () (_ BitVec 8))
(declare-fun b () (_ BitVec 8))
(declare-fun c () (_ BitVec 8))

(assert (bvugt a #x00))
(assert (bvuge b #x00))
(assert (bvugt c #x00))

(assert (bvule a #x09))
(assert (bvule b #x09))
(assert (bvule c #x09))

; slower:
;(assert (= (bvurem (bvadd (bvmul a (_ bv100 8)) (bvmul b (_ bv00 8)) c) #x04) #x00))
;(assert (= (bvurem (bvadd (bvmul c (_ bv100 8)) (bvmul b (_ bv00 8)) a) #x04) #x00))

; faster:
(assert (= (bvand (bvadd (bvmul a (_ bv100 8)) (bvmul b (_ bv00 8)) c) #x03) #x00))
(assert (= (bvand (bvadd (bvmul c (_ bv100 8)) (bvmul b (_ bv00 8)) a) #x03) #x00))

;(check-sat)
;(get-model)
(get-all-models)
;(count-models)
```

Faster version doesn't find remainder, it just isolates two last bits.

8.16 Recreational math, calculator's keypad and divisibility

I've once read about this puzzle. Imagine calculator's keypad:

```
789
456
123
```

If you form any rectangle or square out of keys, like:

```
 7 8 9
+---+
|4 5|6
|1 2|3
+---+
```

The number is 4521. Or 2145, or 5214. All these numbers are divisible by 11, 111 and 1111. One explanation: https://files.eric.ed.gov/fulltext/EJ891796.pdf.

However, I could try to prove that all these numbers are indeed divisible.

```
from z3 import *

"""
We will keep track on numbers using row/col representation:

 |0 1 2 <-col
-|- - -
0|7 8 9
1|4 5 6
2|1 2 3
^
|
row

"""

# map coordinates to number on keypad:
def coords_to_num (r, c):
    return If(And(r==0, c==0), 7,
    If(And(r==0, c==1), 8,
    If(And(r==0, c==2), 9,
    If(And(r==1, c==0), 4,
    If(And(r==1, c==1), 5,
    If(And(r==1, c==2), 6,
    If(And(r==2, c==0), 1,
    If(And(r==2, c==1), 2,
    If(And(r==2, c==2), 3, 9999)))))))))

s=Solver()

# coordinates of upper left corner:
from_r, from_c = Ints('from_r from_c')
# coordinates of bottom right corner:
to_r, to_c = Ints('to_r to_c')

# all coordinates are in [0..2]:
s.add(And(from_r>=0, from_r<=2, from_c>=0, from_c<=2))
s.add(And(to_r>=0, to_r<=2, to_c>=0, to_c<=2))

# bottom-right corner is always under left-upper corner, or equal to it, or to the
   right of it:
s.add(to_r>=from_r)
s.add(to_c>=from_c)

# numbers on keypads for all 4 corners:
LT, RT, BL, BR = Ints('LT RT BL BR')

# ... which are:
s.add(LT==coords_to_num(from_r, from_c))
s.add(RT==coords_to_num(from_r, to_c))
s.add(BL==coords_to_num(to_r, from_c))
s.add(BR==coords_to_num(to_r, to_c))

# 4 possible 4-digit numbers formed by passing by 4 corners:
n1, n2, n3, n4 = Ints('n1 n2 n3 n4')
```

```
s.add(n1==LT*1000 + RT*100 + BR*10 + BL)
s.add(n2==RT*1000 + BR*100 + BL*10 + LT)
s.add(n3==BR*1000 + BL*100 + LT*10 + RT)
s.add(n4==BL*1000 + LT*100 + RT*10 + BR)

# what we're going to do?
prove=False
enumerate_rectangles=True

assert prove != enumerate_rectangles

if prove:
    # prove by finding counterexample.
    # find any variable state for which remainder will be non-zero:
    s.add(And((n1%11) != 0), (n1%111) != 0, (n1%1111) != 0)
    s.add(And((n2%11) != 0), (n2%111) != 0, (n2%1111) != 0)
    s.add(And((n3%11) != 0), (n3%111) != 0, (n3%1111) != 0)
    s.add(And((n4%11) != 0), (n4%111) != 0, (n4%1111) != 0)

    # this is impossible, we're getting unsat here, because no counterexample exist:
    print s.check()

# ... or ...

if enumerate_rectangles:
    # enumerate all possible solutions:
    results=[]
    while True:
        if s.check() == sat:
            m = s.model()
            #print_model(m)
            print m
            print m[n1]
            print m[n2]
            print m[n3]
            print m[n4]
            results.append(m)
            block = []
            for d in m:
                c=d()
                block.append(c != m[d])
            s.add(Or(block))
        else:
            print "results total=", len(results)
            break
```

Enumeration: only 36 rectangles exist on 3*3 keypad:

```
[n1 = 7821,
 BL = 1,
 n2 = 8217,
 to_r = 2,
 LT = 7,
 RT = 8,
 BR = 2,
 n4 = 1782,
 from_r = 0,
 n3 = 2178,
 from_c = 0,
 to_c = 1]
7821
8217
```

```
2178
1782
[n1 = 7931,
 BL = 1,
 n2 = 9317,
 to_r = 2,
 LT = 7,
 RT = 9,
 BR = 3,
 n4 = 1793,
 from_r = 0,
 n3 = 3179,
 from_c = 0,
 to_c = 2]
7931
9317
3179
1793

...

[n1 = 5522,
 BL = 2,
 n2 = 5225,
 to_r = 2,
 LT = 5,
 RT = 5,
 BR = 2,
 n4 = 2552,
 from_r = 1,
 n3 = 2255,
 from_c = 1,
 to_c = 1]
5522
5225
2255
2552
results total= 36
```

8.17 Crossword generator

We assign an integer to each character in crossword, it reflects ASCII code of it.

Then we enumerate all possible horizontal/vertical "sticks" longer than 1 and assign words to them.

For example, there is a horizontal stick of length 3. And we have such 3-letter words in our dictionary: "the", "she", "xor".

We add the following constraint:

```
Or(
        And(chars[X][Y]=='t', chars[X][Y+1]=='h', chars[X][Y+2]=='e'),
        And(chars[X][Y]=='s', chars[X][Y+1]=='h', chars[X][Y+2]=='e'),
        And(chars[X][Y]=='x', chars[X][Y+1]=='o', chars[X][Y+2]=='r'))
```

One of these words would be chosen automatically.

Index of each word is also considered, because duplicates are not allowed.

Sample pattern:

```
**** **********
 * * *  * * * *
***************
 * * *  * * * *
********* *****
```

```
 * * * * * *  *
****** ********
   * * * * *
******** ******
*  * * * * * *
***** *********
* * * *  * * *
***************
* * * *  * * *
********** ****
```

Sample result:

```
spur stimulated
 r e c  i a h e
congratulations
 m u t  a i s c
violation niece
 s a e p e n  n
rector penitent
   i i o c e
accounts herald
s  n g e a r o
press edinburgh
e x e n  t p i
characteristics
t c l r  n e a
satisfying dull

horizontal:
((0, 0), (0, 3)) spur
((0, 5), (0, 14)) stimulated
((2, 0), (2, 14)) congratulations
((4, 0), (4, 8)) violation
((4, 10), (4, 14)) niece
((6, 0), (6, 5)) rector
((6, 7), (6, 14)) penitent
((8, 0), (8, 7)) accounts
((8, 9), (8, 14)) herald
((10, 0), (10, 4)) press
((10, 6), (10, 14)) edinburgh
((12, 0), (12, 14)) characteristics
((14, 0), (14, 9)) satisfying
((14, 11), (14, 14)) dull
vertical:
((8, 0), (14, 0)) aspects
((0, 1), (6, 1)) promise
((10, 2), (14, 2)) exact
((0, 3), (10, 3)) regulations
((10, 4), (14, 4)) seals
((0, 5), (9, 5)) scattering
((10, 6), (14, 6)) entry
((4, 7), (10, 7)) opposed
((0, 8), (4, 8)) milan
((5, 9), (14, 9)) enchanting
((0, 10), (4, 10)) latin
((4, 11), (14, 11)) interrupted
((0, 12), (4, 12)) those
((8, 13), (14, 13)) logical
((0, 14), (6, 14)) descent
```

Unsat is possible if the dictionary is too small or have no words of length present in pattern.

Constructing a crossword puzzle by bruteforce is not feasible, especially in the presence of big dictionary.

The source code:

```
#!/usr/bin/env python3

from z3 import *
import sys

"""
# https://commons.wikimedia.org/wiki/File:Khachbar-1.jpg
pattern=[
"*****",
"* * *",
"* ***",
"*** *",
"* ***",
"* * *",
"*****"]
"""

"""
# https://commons.wikimedia.org/wiki/File:Khachbar-4.jpg
pattern=[
"*******",
"* * * *",
"*******",
"* * * *",
"*******",
"* * * *",
"*******"]
"""

#"""
# https://commons.wikimedia.org/wiki/File:British_crossword.svg
pattern=[
"**** **********",
" * * *  * * * *",
"***************",
" * * *  * * * *",
"********* *****",
" * * * * * *  *",
"****** ********",
"   * * * * *   ",
"******** ******",
"*  * * * * * * ",
"********** *****",
"* * * *  * * * ",
"***************",
"* * * *  * * * ",
"********** ****"]
#"""

HEIGHT=len(pattern)
WIDTH=len(pattern[0])

# scan pattern[] and find all "sticks" longer than 1 and collect its coordinates:

horizontal=[]

in_the_middle=False
for r in range(HEIGHT):
    for c in range(WIDTH):
        if pattern[r][c]=='*' and in_the_middle==False:
            in_the_middle=True
            start=(r,c)
        elif pattern[r][c]==' ' and in_the_middle==True:
            if c-start[1]>1:
                horizontal.append ((start, (r, c-1)))
            in_the_middle=False
    if in_the_middle:
        if c-start[1]>1:
            horizontal.append ((start, (r, c)))
        in_the_middle=False

vertical=[]

in_the_middle=False
for c in range(WIDTH):
    for r in range(HEIGHT):
        if pattern[r][c]=='*' and in_the_middle==False:
            in_the_middle=True
            start=(r,c)
        elif pattern[r][c]==' ' and in_the_middle==True:
            if r-start[0]>1:
                vertical.append ((start, (r-1, c)))
            in_the_middle=False
    if in_the_middle:
        if r-start[0]>1:
            vertical.append ((start, (r, c)))
        in_the_middle=False

# for the first simple pattern, we will have such coordinates of "sticks":
# horizontal=[((0, 0), (0, 4)), ((2, 2), (2, 4)), ((3, 0), (3, 2)), ((4, 2), (4, 4)), ((6, 0), (6, 4))]
# vertical=[((0, 0), (6, 0)), ((0, 2), (6, 2)), ((0, 4), (6, 4))]

# the list in this file is assumed to not have duplicates, otherwise duplicates can be present in the final resulting crossword:
with open("words.txt") as f:
    content = f.readlines()
words = [x.strip() for x in content]

# FIXME: slow, called too often
def find_words_len(l):
    rt=[]
    i=0
    for word in words:
        if len(word)==l:
            rt.append ((i, word))
        i=i+1
    return rt

# 2D array of ASCII codes of all characters:
chars=[[Int('chars_%d_%d' % (r, c)) for c in range(WIDTH)] for r in range(HEIGHT)]
# indices of horizontal words:
horizontal_idx=[Int('horizontal_idx_%d' % i) for i in range(len(horizontal))]
# indices of vertical words:
vertical_idx=[Int('vertical_idx_%d' % i) for i in range(len(vertical))]

s=Solver()
set_param("parallel.enable", True)
```

```
# this function takes coordinates, word length and word itself
# for "hello", it returns array like:
# [chars[0][0]==ord('h'), chars[0][1]==ord('e'), chars[0][2]==ord('l'), chars[0][3]==
    ord('l'), chars[0][4]==ord('o')]
def form_H_expr(r, c, l, w):
    return [chars[r][c+i]==ord(w[i]) for i in range(l)]

# now we find all horizonal "sticks", we find all possible words of corresponding
    length...
for i in range(len(horizontal)):
    h=horizontal[i]
    _from=h[0]
    _to=h[1]
    l=_to[1]-_from[1]+1
    list_of_ANDs=[]
    for idx, word in find_words_len(l):
        # at this point, we form expression like:
        # And(chars[0][0]==ord('h'), chars[0][1]==ord('e'), chars[0][2]==ord('l'),
            chars[0][3]==ord('l'), chars[0][4]==ord('o'), horizontal_idx[]==...)
        list_of_ANDs.append(And(form_H_expr(_from[0], _from[1], l, word)+[
            horizontal_idx[i]==idx]))
    # at this point, we form expression like:
    # Or(And(chars...==word1), And(chars...==word2), And(chars...==word3))
    s.add(Or(*list_of_ANDs))

# same for vertical "sticks":
def form_V_expr(r, c, l, w):
    return [chars[r+i][c]==ord(w[i]) for i in range(l)]

for i in range(len(vertical)):
    v=vertical[i]
    _from=v[0]
    _to=v[1]
    l=_to[0]-_from[0]+1
    list_of_ANDs=[]
    for idx, word in find_words_len(l):
        list_of_ANDs.append(And(form_V_expr(_from[0], _from[1], l, word)+[
            vertical_idx[i]==idx]))
    s.add(Or(*list_of_ANDs))

# we collected indices of horizonal/vertical words to make sure they will not be
    duplicated on resulting crossword:
s.add(Distinct(*(horizontal_idx+vertical_idx)))

def print_model (m):
    print ("")
    for r in range(HEIGHT):
        for c in range(WIDTH):
            if pattern[r][c]=='*':
                sys.stdout.write(chr(m[chars[r][c]].as_long()))
            else:
                sys.stdout.write(' ')
        print ("")
    print ("")

    print ("horizontal:")
    for i in range(len(horizontal)):
        print (horizontal[i], words[m[horizontal_idx[i]].as_long()])

    print ("vertical:")
    for i in range(len(vertical)):
        print (vertical[i], words[m[vertical_idx[i]].as_long()])

N=10
# get 10 results:
results=[]
for i in range(N):
    if s.check() == sat:
        m = s.model()
        print_model(m)
        results.append(m)
        block = []
        for d in m:
            c=d()
            block.append(c != m[d])
        s.add(Or(block))
    else:
        print ("total results", len(results))
        break
```

The files, including my dictionary: https://smt.st/current_tree/puzzles/cross.

Further work: rewrite it to SAT, because it's too slow.

8.18 Almost recreational math: missing operation(s) puzzle

The equation:

```
1 ? 2 ? 3 ? 4 ? 5 ? 6 ? 7 ? 8 ? 9 = 0
```

Fill ? with -, + or * operation, and find such a sequence, so that the equation would be true. This is tricky, because of operator precedence, multiplication must be handled at first. Brackets are also possible.

See also: https://www.dcode.fr/missing-operators-equation-solver, https://brilliant.org/wiki/arithmetic-puzzles-operator-search/.

To solve this, I just enumerate all possible ordered binary search tree of 9 elements, which are almost the same as expression with 9 terms and all possible variations of brackets.

Then we try each expression...

```
#!/usr/bin/env python3
from z3 import *

CONST=1234
TOTAL=9

# prepare a list in form: [(1,"1"),(2,"2"),(3,"3")...]
# rationale: enum_ordered() yields both expression tree and expression as a string...
input_values=[]
for i in range(TOTAL):
    input_values.append((i+1, str(i+1)))

OPS=TOTAL-1

ops=[Int('op_%d' % i) for i in range(OPS)]

# this is a hack... operation number. resetting after each tree:
n=-1

# select operation...
def op(l, r, n):
    return If(ops[n]==0, l+r,
        If(ops[n]==1, l-r,
        If(ops[n]==2, l*r,
        0))) # default
```

```
# enumerate all possible ordered binary search trees
# copypasted from https://stackoverflow.com/questions/14900693/enumerate-all-full-labeled-binary-tree
# this generator yields both expression and string

# expression may have form like (please note, many constants are optimized/folded):

# If(op_1 == 0,
#    1 +
#    If(op_0 == 0, 5, If(op_0 == 1, -1, If(op_0 == 2, 6, 0))),
#    If(op_1 == 1,
#       1 -
#       If(op_0 == 0,
#          5,
#          If(op_0 == 1, -1, If(op_0 == 2, 6, 0))),
#       If(op_1 == 2,
#          1*
#          If(op_0 == 0,
#             5,
#             If(op_0 == 1, -1, If(op_0 == 2, 6, 0))),
#          0)))

# string is like "(1 op1 (2 op0 3))", opX substring will be replaced by operation name after (-, +, *)

def enum_ordered(labels):
    global n
    if len(labels) == 1:
        yield (labels[0][0], labels[0][1])
    else:
        for i in range(1, len(labels)):
            for left in enum_ordered(labels[:i]):
                for right in enum_ordered(labels[i:]):
                    n=n+1
                    yield (op(left[0], right[0], n), "("+left[1]+" op"+str(n)+" "+right[1]+")")

for tree in enum_ordered(input_values):
    s=Solver()
    # operations in 0..2 range...
    for i in range(OPS):
        s.add(And(ops[i]>=0, ops[i]<=2))
    s.add(tree[0]==CONST)
    if s.check()==sat:
        m=s.model()
        tmp=tree[1]
        for i in range(OPS):
            op_s=["+", "-", "*"][m[ops[i]].as_long()]
            tmp=tmp.replace("op"+str(i), op_s)
        print (tmp, "=", eval(tmp))
    n=-1
```

For 9 terms, there are 1430 binary trees, or expressions (9th *Catalan number*).

For 0, there are 1391 solutions, some of them:

```
(1 + (2 + (3 + (4 + (5 * (6 - (7 - (8 - 9)))))))) = 0
(1 + (2 + (3 + (4 + (5 * (6 - ((7 - 8) + 9))))))) = 0
(1 + (2 + (3 + (4 + (5 * ((6 - 7) + (8 - 9))))))) = 0
(1 + (2 + (3 - (4 + (5 - ((6 * (7 - 8)) + 9)))))) = 0
(1 + (2 + (3 + (4 + (5 * (((6 - 7) + 8) - 9)))))) = 0
(1 - (2 + (3 + (4 + ((5 - 6) + (7 * (8 - 9))))))) = 0
(1 - (2 + (3 + (4 + ((5 - 6) * ((7 - 8) + 9)))))) = 0
```

```
...

((((((1 - (2 - 3)) * (4 - 5)) - 6) + 7) - 8) + 9) = 0
(((((((1 - 2) + 3) + (4 - 5)) * 6) - 7) - 8) + 9) = 0
((((((1 + (2 * (3 * 4))) + 5) - 6) - 7) - 8) - 9) = 0
((((((1 * ((2 + 3) + 4)) - 5) + 6) + 7) - 8) - 9) = 0
(((((((1 + 2) - (3 + 4)) + 5) + 6) - 7) * 8) * 9) = 0
(((((((1 * (2 - 3)) * 4) + 5) * 6) - 7) - 8) + 9) = 0
((((((((1 - 2) + 3) + 4) * 5) - 6) - 7) - 8) - 9) = 0
```

There are no solutions for 5432. But for 5430:

```
(1 + (2 + (3 * ((4 * (5 * (6 * (7 + 8)))) + 9)))) = 5430
(1 + (2 + (3 * ((4 * ((5 * 6) * (7 + 8))) + 9)))) = 5430
(1 + (2 + (3 * (((4 * 5) * (6 * (7 + 8))) + 9)))) = 5430
(1 + (2 + (3 * (((4 * (5 * 6)) * (7 + 8)) + 9)))) = 5430
(1 + (2 + (3 * ((((4 * 5) * 6) * (7 + 8)) + 9)))) = 5430
((1 + 2) + (3 * ((4 * (5 * (6 * (7 + 8)))) + 9))) = 5430
```

Surely, several of these expressions are equivalent to each other, due to associative property of multiplication and addition.

For 1234:

```
(1 * (2 * (((3 - (4 - ((5 + 6) * 7))) * 8) + 9))) = 1234
(1 + (2 + ((((3 * ((4 + 5) * 6)) - 7) * 8) - 9))) = 1234
(1 + (2 + (((((3 * (4 + 5)) * 6) - 7) * 8) - 9))) = 1234
(1 + ((2 + (3 * ((4 + 5) * (6 + (7 - 8))))) * 9)) = 1234
(1 + ((2 + (3 * ((4 + 5) * ((6 + 7) - 8)))) * 9)) = 1234
(1 + ((2 + (3 * ((4 + (5 - 6)) * (7 + 8)))) * 9)) = 1234
(1 + ((2 + (3 * (((4 + 5) - 6) * (7 + 8)))) * 9)) = 1234
(1 + ((2 + (3 * (((4 * (5 + 6)) - 7) + 8))) * 9)) = 1234
(1 + ((2 + ((3 * (4 + 5)) * (6 + (7 - 8)))) * 9)) = 1234
(1 - ((2 - (((3 + 4) * 5) + ((6 + 7) * 8))) * 9)) = 1234
(1 + ((2 + ((3 * (4 + (5 - 6))) * (7 + 8))) * 9)) = 1234
(1 + ((2 + ((3 * ((4 + 5) - 6)) * (7 + 8))) * 9)) = 1234
(1 + ((2 - (((3 * (4 - 5)) - 6) * (7 + 8))) * 9)) = 1234
(1 - ((2 + ((3 * ((4 - (5 + 6)) * 7)) + 8)) * 9)) = 1234
(1 - ((2 + ((3 * (((4 - 5) - 6) * 7)) + 8)) * 9)) = 1234
(1 - ((2 + (((3 * (4 - (5 + 6))) * 7) + 8)) * 9)) = 1234
(1 - ((2 + (((3 * ((4 - 5) - 6)) * 7) + 8)) * 9)) = 1234
(1 - ((2 - ((((3 * (4 + 5)) - 6) * 7) - 8)) * 9)) = 1234
(1 - (((2 - ((3 + 4) * 5)) - ((6 + 7) * 8)) * 9)) = 1234
(1 - (((2 - ((3 + (4 * 5)) * 6)) + (7 - 8)) * 9)) = 1234
(1 + (((2 + (3 - (4 * (5 - (6 * 7))))) * 8) + 9)) = 1234
(1 + (((2 + (3 + (4 * ((5 * 6) + 7)))) * 8) + 9)) = 1234
(1 - (((2 + (3 * ((4 - (5 + 6)) * 7))) + 8) * 9)) = 1234
(1 - (((2 + (3 * (((4 - 5) - 6) * 7))) + 8) * 9)) = 1234
(1 - (((2 + ((3 * (4 - (5 + 6))) * 7)) + 8) * 9)) = 1234
(1 - (((2 + ((3 * ((4 - 5) - 6)) * 7)) + 8) * 9)) = 1234
(1 - (((2 - (((3 + (4 * 5)) * 6) - 7)) - 8) * 9)) = 1234
(1 + ((((2 + 3) - (4 * (5 - (6 * 7)))) * 8) + 9)) = 1234
(1 - ((((2 - (3 * ((4 + 5) * 6))) + 7) * 8) - 9)) = 1234
(1 - ((((2 - ((3 + (4 * 5)) * 6)) + 7) - 8) * 9)) = 1234
((1 + 2) + ((((3 * ((4 + 5) * 6)) - 7) * 8) - 9)) = 1234
((1 + 2) + (((((3 * (4 + 5)) * 6) - 7) * 8) - 9)) = 1234
((1 + (2 + (((3 * ((4 + 5) * 6)) - 7) * 8))) - 9) = 1234
((1 + (2 + ((((3 * (4 + 5)) * 6) - 7) * 8))) - 9) = 1234
((1 + ((2 + (3 - (4 * (5 - (6 * 7))))) * 8)) + 9) = 1234
((1 + ((2 + (3 + (4 * ((5 * 6) + 7)))) * 8)) + 9) = 1234
((1 - ((2 - ((3 * ((4 + 5) * 6)) - 7)) * 8)) + 9) = 1234
((1 - ((2 - (((3 * (4 + 5)) * 6) - 7)) * 8)) + 9) = 1234
((1 + (((2 + 3) - (4 * (5 - (6 * 7)))) * 8)) + 9) = 1234
```

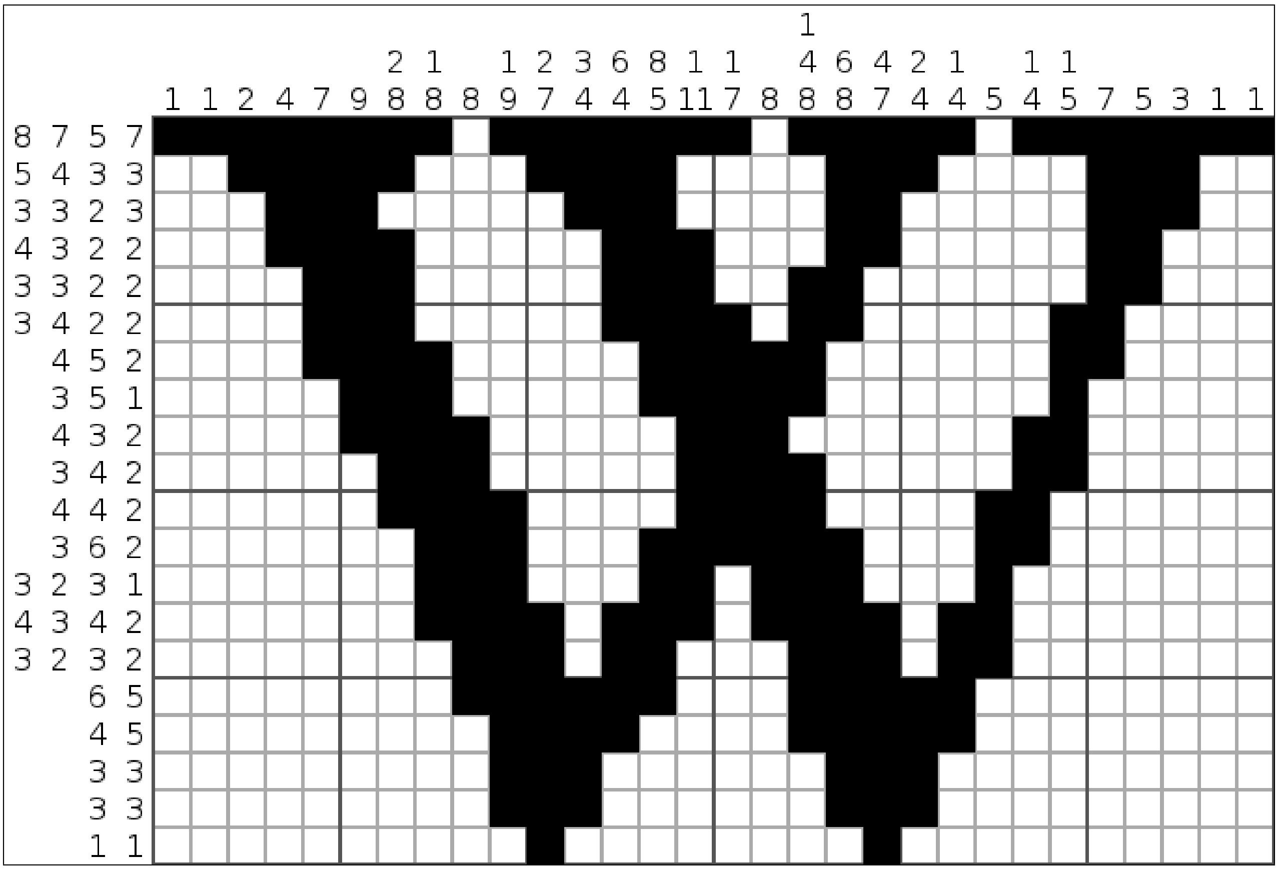
```
((1 - (((2 - (3 * ((4 + 5) * 6))) + 7) * 8)) + 9) = 1234
((1 - (((2 - ((3 * (4 + 5)) * 6)) + 7) * 8)) + 9) = 1234
(((1 + 2) + (((3 * ((4 + 5) * 6)) - 7) * 8)) - 9) = 1234
(((1 + 2) + ((((3 * (4 + 5)) * 6) - 7) * 8)) - 9) = 1234
```

The problem is easy enough to be solved using my toy-level MK85 SMT-solver:

```
#!/usr/bin/env python3
from MK85 import *

n=-1

CONST=13

#TOTAL=9
TOTAL=7

BIT_WIDTH=8

s=MK85(verbose=0)

input_values=[]
for i in range(TOTAL):
    input_values.append((s.BitVecConst(i+1, BIT_WIDTH), str(i+1)))

OPS=TOTAL-1

ops=[s.BitVec('op_%d' % i,2) for i in range(OPS)]

for i in range(OPS):
    s.add(And(ops[i]>=0, ops[i]<=2))

def op(l, r, n):
    return s.If(ops[n]==s.BitVecConst(0, 2), l+r,
        s.If(ops[n]==s.BitVecConst(1, 2), l-r,
        s.If(ops[n]==s.BitVecConst(2, 2), l*r,
            s.BitVecConst(0, BIT_WIDTH))))

def enum_ordered(labels):
    global n
    if len(labels) == 1:
        yield (labels[0][0], labels[0][1])
    else:
        for i in range(1, len(labels)):
            for left in enum_ordered(labels[:i]):
                for right in enum_ordered(labels[i:]):
                    n=n+1
                    yield (op(left[0], right[0], n), "("+left[1]+" op"+str(n)+" "+
                        right[1]+")")

for tree in enum_ordered(input_values):
    s.add(tree[0]==CONST)

    for i in range(OPS):
        s.add(ops[i]!=3)

    if s.check():
        m=s.model()
        #print ("sat", tree[1])
        tmp=tree[1]
        for i in range(OPS):
            op_s=["+", "-", "*"][m['op_%d' % i]]
            tmp=tmp.replace("op"+str(i), op_s)
        print (tmp, "=", eval(tmp))
        # show only first solution...
        exit(0)
```

8.19 Nonogram puzzle solver

This is a sample one, from Wikipedia:



```
#!/usr/bin/env python3
from z3 import *

# https://ocaml.org/learn/tutorials/99problems.html
#rows=[[3],[2,1],[3,2],[2,2],[6],[1,5],[6],[1],[2]]
#cols=[[1,2],[3,1],[1,5],[7,1],[5],[3],[4],[3]]

# https://ocaml.org/learn/tutorials/99problems.html
#rows= [[14], [1,1], [7,1], [3,3], [2,3,2], [2,3,2], [1,3,6,1,1], [1,8,2,1],
    [1,4,6,1], [1,3,2,5,1,1], [1,5,1], [2,2], [2,1,1,1,2], [6,5,3], [12]]
#cols= [[7], [2,2], [2,2], [2,1,1,1,1], [1,2,4,2], [1,1,4,2], [1,1,2,3], [1,1,3,2],
    [1,1,1,2,2,1], [1,1,5,1,2], [1,1,7,2], [1,6,3], [1,1,3,2], [1,4,3], [1,3,1],
    [1,2,2], [2,1,1,1,1], [2,2], [2,2], [7]]

# https://en.wikipedia.org/wiki/File:Nonogram_wiki.svg
rows=[[8,7,5,7], [5,4,3,3], [3,3,2,3], [4,3,2,2], [3,3,2,2], [3,4,2,2], [4,5,2],
    [3,5,1], [4,3,2], [3,4,2], [4,4,2], [3,6,2], [3,2,3,1], [4,3,4,2], [3,2,3,2],
    [6,5], [4,5], [3,3], [3,3], [1,1]]
cols=[[1], [1], [2], [4], [7], [9], [2,8], [1,8], [8], [1,9], [2,7], [3,4], [6,4],
    [8,5], [1,11], [1,7], [8], [1,4,8], [6,8], [4,7], [2,4], [1,4], [5], [1,4], [1,5],
     [7], [5], [3], [1], [1]]
```

```
# http://puzzlygame.com/nonogram/11/
#rows=[[12], [1,2,1,2], [6,6], [1,1,1,2], [6,6], [3,2,1,2], [5,2,6], [2,3,2,1,2],
    [3,3,2,1,2], [3,4,2,1,2], [3,2,2,1,2], [7,2,1,2], [7,2,1,2], [5,2,6], [5,2,1,2],
    [3,2,6], [6,1,2], [1,1,6], [6,1,2], [12]]
#cols=[[5], [9], [11], [4,7], [2,6,6,1], [1,1,1,4,4,1,2], [1,1,2,6,2,2],
    [1,1,3,2,4,2], [1,3,3,4,4], [3,6,2], [2,2,1], [1,1], [3,3], [1,4,4,1],
    [1,1,1,8,1,1,1], [1,1,1,1,1,1,1,1], [3,1,1,1,1,3], [6,6], [14], [8]]

# http://puzzlygame.com/nonogram/24/
#rows=[[4], [1,8,1], [3,11,3], [3,14,4], [4,22], [1,4,22], [28], [27], [3,11,8],
    [9,9], [8,8], [7,8], [2,1,7], [2,6], [3,5], [2,4], [4], [2], [2], [2], [2], [2],
    [2], [2], [2]]
#cols=[[2], [3], [4], [1,2], [4], [5], [7], [3,5], [1,5], [7], [7], [9], [11],
    [10,2], [11,2], [12,2,2], [9,3], [8,4,2], [15,2], [17,2], [19], [18], [15], [12],
    [9], [7], [4], [3], [3], [2], [3], [2], [3], [2], [2]]

# http://puzzlygame.com/nonogram/792/
#rows=[[2],[4],[2,2],[2,1],[1,1,2,2],[1,4,2,2],[2,1,1,2],[4,2,2],[1,3,2,2,2,4,4],
    [1,1,1,2,4,2,1,1,1,1],[1,1,5,2,2,2,1,1,1,1],[2,2,1,1,2,2,2,4,4],
    [1,2,3,2,3],[2,1,2,26],[2,1,2,2,1],[4,2,1,4,1,4],[1,2,1,1,1,1,1,5,4],
    [2,1,6,1,1,1,1,4],[2,1,2,2,1,5,4],[2,1,1,1,1,1,1,1,1,1,2],[2,1,1,1,1,5,4],
    [2,1,8,1,4],[2,1,1,1,1,4],[2,1,1,1,1,4],[35]]
#cols=[[1,2,3,1,1,1],[1,2,1,1,3,1],[4,1,2,10],[2,3,2,11],
    [2,1,2,1,1,1],[1,1,2,1,1,1,1],[3,6,1],[3,2,1,1],
    [3,1,1],[12],[2,1],[3,4,1],[2,1,3,4],[2,1,1,1,1,1,1],
    [2,1,1,1,1,1],[2,1,1,1,1,1],[2,1,1,1,1,1,1],[2,1,3,4],[3,4,1],
    [2,1],[14],[2,1,1],[2,1,1],[2,1,5,1],[2,1,1,1,1,1],[2,1,5,1],
    [2,4,1,1,1,1,1],[2,1,1,1,5,1],[2,1,1,1,1],
    [2,4,1,1],[2,1,1],[2,2,4,1,10],[2,2,1,1,1,4,5],[2,1,1,1,10],[2,4,1,10]]

# http://puzzlygame.com/nonogram/561/
#rows=[[13,1,1,1,13], [1,2,1,1,2,2,2,1,2,1,1,2,1], [1,9,2,2,1,9,1], [35], [1,1,1,1],
    [1,1,3,3,3,3,3,2,1,1], [5,2,3,3,3,3,4,5], [1,1], [3,3,3,3], [1,3,2,17,2,3,1],
    [1,2,17,2,1], [1,3,2,1,1,1,1,2,3,1], [1,3,2,1,1,9,1,1,2,3,1],
    [1,3,2,3,1,1,1,3,2,3,1], [1,3,2,3,1,1,1,3,2,3,1], [1,3,2,2,1,1,1,3,3,1],
    [1,3,2,2,1,1,10,1], [1,3,1,9,1,3,3,1], [1,3,2,1,2,1,10,1], [1,2,3,1,3,5,4,1],
    [1,18,4,3,1], [1,20,3,2,1], [1,20,4,3,1], [1,1,1,1,1,1], [4,3,6,3,15]]
#cols=[[7,15,1], [1,1,1,1,1], [1,1,1,2,9,2,1], [1,1,1,1,9,2,1], [7,2,8,3], [4,1,5],
    [1,2,12,4,1], [4,2,8,7], [1,2,1,6], [4,2,7,1,3,1], [1,2,1,2,3,1,3,1],
    [4,2,6,2,3,1], [4,1,2,7,1], [1,2,2,4,1,3,1], [4,1,2,1,1,3,1], [3,2,2,1,1,3],
    [1,1,2,1,8], [4,2,2,5,5,1], [3,1,2,1,6], [1,2,2,1,3], [4,1,2,1,3,1],[1,2,2,7,3,1],
     [4,1,2,1], [4,2,14,1], [1,2,1,2,10,1], [4,2,14,1], [1,2,1,1,3,3], [4,2,6,1,2,1],
    [1,2,10,1,1,1], [4,1,1,2,1], [7,2,10,3], [1,1,1,1,12,1], [1,1,1,2,12,1],
    [1,1,1,1,1], [7,15,1]]

WIDTH=len(cols)
HEIGHT=len(rows)

s=Solver()

# part I, for all rows:
row_islands=[[BitVec('row_islands_%d_%d' % (j, i), WIDTH) for i in range(len(rows[j])
    )] for j in range(HEIGHT)]
row_island_shift=[[BitVec('row_island_shift_%d_%d' % (j, i), WIDTH) for i in range(
    len(rows[j]))] for j in range(HEIGHT)]
# this is a bitvector representing final image, for all rows:
row_merged_islands=[BitVec('row_merged_islands_%d' % j, WIDTH) for j in range(HEIGHT)
    ]

for j in range(HEIGHT):
    q=rows[j]
    for i in range(len(q)):
        s.add(row_island_shift[j][i] >= 0)
        s.add(row_island_shift[j][i] <= WIDTH-q[i])
        s.add(row_islands[j][i]==(2**q[i]-1) << row_island_shift[j][i])

    # must be an empty cell(s) between islands:
    for i in range(len(q)-1):
        s.add(row_island_shift[j][i+1] > row_island_shift[j][i]+q[i])

    s.add(row_island_shift[j][len(q)-1]<WIDTH)

    # OR all islands into one:
    expr=row_islands[j][0]
    for i in range(len(q)-1):
        expr=expr | row_islands[j][i+1]
    s.add(row_merged_islands[j]==expr)

# similar part, for all columns:
col_islands=[[BitVec('col_islands_%d_%d' % (j, i), HEIGHT) for i in range(len(cols[j
   ]))] for j in range(WIDTH)]
col_island_shift=[[BitVec('col_island_shift_%d_%d' % (j, i), HEIGHT) for i in range(
   len(cols[j]))] for j in range(WIDTH)]
# this is a bitvector representing final image, for all columns:
col_merged_islands=[BitVec('col_merged_islands_%d' % j, HEIGHT) for j in range(WIDTH)
   ]

for j in range(WIDTH):
    q=cols[j]
    for i in range(len(q)):
        s.add(col_island_shift[j][i] >= 0)
        s.add(col_island_shift[j][i] <= HEIGHT-q[i])
        s.add(col_islands[j][i]==(2**q[i]-1) << col_island_shift[j][i])

    # must be an empty cell(s) between islands:
    for i in range(len(q)-1):
        s.add(col_island_shift[j][i+1] > col_island_shift[j][i]+q[i])

    s.add(col_island_shift[j][len(q)-1]<HEIGHT)

    # OR all islands into one:
    expr=col_islands[j][0]
    for i in range(len(q)-1):
        expr=expr | col_islands[j][i+1]
    s.add(col_merged_islands[j]==expr)

#
   ----------------------------------------------------------------------------------------

# make "merged" vectors equal to each other:
for r in range(HEIGHT):
    for c in range(WIDTH):
        # lowest bits must be equal to each other:
        s.add(Extract(0,0,row_merged_islands[r]>>c) == Extract(0,0,col_merged_islands
            [c]>>r))

def print_model(m):
    for r in range(HEIGHT):
        rt=""
        for c in range(WIDTH):
            if (m[row_merged_islands[r]].as_long()>>c)&1==1:
                rt=rt+"*"
            else:
                rt=rt+" "
        print (rt)

print (s.check())
m=s.model()
print_model(m)
exit(0)

# ... or ...

# enumerate all solutions (it's usually a single one):
results=[]
while s.check() == sat:
    m = s.model()
    print_model(m)

    results.append(m)
    block = []
    for d in m:
        t=d()
        block.append(t != m[d])
    s.add(Or(block))

print ("results total=", len(results))
```

The result:

```
sat
******** ******* ***** *******
  *****   ****    ***    ***
   ***     ***    **     ***
   ****     ***   **     **
    ***     ***  **      **
    ***     **** **     **
    ****     *****      **
     ***     *****      *
     ****     ***      **
      ***     ****     **
      ****    ****    **
       ***   ******   **
       ***   ** ***   *
       **** *** **** **
        *** **   *** **
        ******   *****
         ****    *****
         ***      ***
         ***      ***
          *        *
```

How it works (briefly). Given a row of width 8 and input (or *clue*) like [3,2], we create two *islands* of two bitstrings of corresponding lengths:

```
00000111
00000011
```

The whole length of each bitvector/bitstring is 8 (width of row).

We then define another variable: `island_shift`, for each *island*, which defines a count, on which a bitstring is shifted left. We also calculate limits for each *island*: position of each one must not be lower/equal then the position of the previous *island*.

All islands are then merged into one (`merged_islands[]`) using OR operation:

```
11100000
00001100

->

11101100
```

merged_islands[] is a final representation of row — how it will be printed.

Now repeat this all for all rows and all columns.

The final step: make corresponding bits in XXX_merged_islands[] of each row and column to be equal to each other. In other words, col_merged_islands[] must be equal to row_merged_islands[], but rotated by 90°.

The solver is surprisingly fast even on hard puzzles.

Further work: colored nonograms.

8.20 ”Feed the kids” puzzle

```
There is a basket containing an apple, a banana, a cherry and a date.
Four children named Erica, Frank, Greg and Hank are each to be given a piece of the
    fruit.
Erica likes cherries and dates;
Frank likes apples and cherries;
Greg likes bananas and cherries;
and Hank likes apples, bananas, and dates.
Figure 1.1.1 describes the situation.
The problem is to give each child a piece of fruit that he or she likes.
```

(von Nora Hartsfield, Gerhard Ringel – Pearls in Graph Theory: A Comprehensive Introduction)

There are only 3 ways to allocate food to kids. The problem is also small enough to be solved using my toy-level MK85 SMT-solver...

```
(set-logic QF_BV)
(set-info :smt-lib-version 2.0)

(declare-fun E () (_ BitVec 2))
(declare-fun F () (_ BitVec 2))
(declare-fun G () (_ BitVec 2))
(declare-fun H () (_ BitVec 2))

; apples = 0
; bananas = 1
; cherries = 2
; dates = 3

; children's preferences:

(assert
        (or
                (= E (_ bv2 2))
                (= E (_ bv3 2))
        )
)

(assert
        (or
                (= F (_ bv0 2))
                (= F (_ bv2 2))
        )
)

(assert
        (or
                (= G (_ bv1 2))
                (= G (_ bv2 2))
        )
)

(assert
        (or
                (= H (_ bv0 2))
                (= H (_ bv2 2))
                (= H (_ bv3 2))
        )
)

; each child must get a food of one type:

(assert (distinct E F G H))

; enumerate all possible solutions:

(get-all-models)

;(model
;        (define-fun E () (_ BitVec 2) (_ bv2 2)) ; 0x2
;        (define-fun F () (_ BitVec 2) (_ bv0 2)) ; 0x0
;        (define-fun G () (_ BitVec 2) (_ bv1 2)) ; 0x1
;        (define-fun H () (_ BitVec 2) (_ bv3 2)) ; 0x3
;)
;(model
;        (define-fun E () (_ BitVec 2) (_ bv3 2)) ; 0x3
;        (define-fun F () (_ BitVec 2) (_ bv2 2)) ; 0x2
;        (define-fun G () (_ BitVec 2) (_ bv1 2)) ; 0x1
;        (define-fun H () (_ BitVec 2) (_ bv0 2)) ; 0x0
;)
;(model
;        (define-fun E () (_ BitVec 2) (_ bv3 2)) ; 0x3
;        (define-fun F () (_ BitVec 2) (_ bv0 2)) ; 0x0
;        (define-fun G () (_ BitVec 2) (_ bv1 2)) ; 0x1
;        (define-fun H () (_ BitVec 2) (_ bv2 2)) ; 0x2
;)
;Model count: 3
```

```
#!/usr/bin/env python3
from z3 import *

# apples = 0
# bananas = 1
# cherries = 2
# dates = 3

E, F, G, H = Ints('E F G H')

s=Solver()

# children's preferences:
s.add(Or(E==2, E==3))
s.add(Or(F==0, F==2))
s.add(Or(G==1, G==2))
s.add(Or(H==0, H==1, H==3))

# each child must get a food of one type:
```

```
s.add(Distinct(E,F,G,H))

# enumerate all possible solutions:
results=[]
while True:
    if s.check() == sat:
        m = s.model()
        print (m)

        results.append(m)
        block = []
        for d in m:
            c=d()
            block.append(c != m[d])
        s.add(Or(block))
    else:
        print ("results total=", len(results))
        break
"""
There are only 3 ways to allocate food to kids:

[G = 1, F = 2, E = 3, H = 0]
[G = 2, F = 0, E = 3, H = 1]
[G = 1, F = 0, E = 2, H = 3]
results total= 3
"""
```

8.21 CSPLIB problem 018

https://github.com/csplib/csplib/blob/master/Problems/prob018/specification.md:

```
You are given an 8 pint bucket of water, and two empty buckets which can contain 5
   and 3 pints respectively.
You are required to divide the water into two by pouring water between buckets (that
   is, to end up with
4 pints in the 8 pint bucket, and 4 pints in the 5 pint bucket).

What is the minimum number of transfers of water between buckets? The challenge is to
    solve this as
a planning problem (encoded into satisfiability or constraint satisfaction) with an
   efficiency approaching
(or exceeding) a simple enumeration.
```

I've reworked the Prolog code (https://github.com/csplib/csplib/blob/master/Problems/prob018/models/enumerate.pl) into this...

For another solution, see also: https://github.com/YosysHQ/SymbiYosys/blob/master/docs/examples/puzzles/pour_853_to_4.sv.

```
from z3 import *

def _try (POURINGS):
    print "* try %d" % POURINGS
    STATES=POURINGS+1

    A=[Int('A_%d' % i) for i in range(STATES)]
    B=[Int('B_%d' % i) for i in range(STATES)]
    C=[Int('C_%d' % i) for i in range(STATES)]
    op=[Int('op_%d' % i) for i in range(STATES)]

    def Z3_min(a,b):
        return If(a<b, a, b)
    s=Solver()

    # volumes (not states):
    jug_A, jug_B, jug_C = 8,5,3

    # "columns": If(And(op==..., preconditions), And(what next state will have), ...)
    for cur in range(STATES-1):
        next=cur+1
        s.add(If(And(op[cur]==0, A[cur]>0, B[cur]<jug_B), And(B[next]==Z3_min(jug_B,
           B[cur]+A[cur]), A[next]==A[cur]-(B[next]-B[cur]), C[next]==C[cur]),
            If(And(op[cur]==1, A[cur]>0, C[cur]<jug_C), And(C[next]==Z3_min(jug_C, C[
               cur]+A[cur]), A[next]==A[cur]-(C[next]-C[cur]), B[next]==B[cur]),
            If(And(op[cur]==2, B[cur]>0), And(A[next]==A[cur]+B[cur], C[next]==C[cur
               ], B[next]==0),
            If(And(op[cur]==3, B[cur]>0, C[cur]<jug_C), And(A[next]==A[cur], C[next
               ]==Z3_min(jug_C, C[cur]+B[cur]), B[next]==B[cur]-(C[next]-C[cur])),
            If(And(op[cur]==4, C[cur]>0), And(A[next]==A[cur]+C[cur], B[next]==B[cur
               ], C[next]==0),
            If(And(op[cur]==5, C[cur]>0), And(A[next]==A[cur], B[next]==Z3_min(jug_B,
                B[cur]+C[cur]), C[next]==C[cur]-(B[next]-B[cur])),
            False)))))))

    # no state must repeat:
    for i in range(STATES):
        for j in range(i):
            s.add(Or(A[i]!=A[j], B[i]!=B[j], C[i]!=C[j]))

    # initial and final state:
    s.add(And(A[0]==8, B[0]==0, C[0]==0))
    s.add(And(A[STATES-1]==4, B[STATES-1]==4, C[STATES-1]==0))

    if s.check()==unsat:
        return

    m=s.model()

    ops_names=["A->B", "A->C", "B->A", "B->C", "C->A", "C->B"]
    for i in range(STATES):
        print "state %d, %d-%d-%d" % (i, m[A[i]].as_long(), m[B[i]].as_long(), m[C[i
           ]].as_long())
        if i!=STATES-1:
            print "op_%d = %s" % (i, ops_names[m[op[i]].as_long()])
    exit(0)

#_try(7)
#exit(0)

for i in range(20):
    _try(i)
```

```
state 0, 8-0-0
op_0 = A->B
state 1, 3-5-0
op_1 = B->C
state 2, 3-2-3
op_2 = C->A
state 3, 6-2-0
op_3 = B->C
state 4, 6-0-2
op_4 = A->B
```

```
state 5, 1-5-2
op_5 = B->C
state 6, 1-4-3
op_6 = C->A
state 7, 4-4-0
```

8.22 Something else

SAT-based solver for the Hexiom logic puzzle

Couple puzzles in F# for Z3

Chapter 9

Graph coloring

9.1 Map coloring

It's possible to color all countries on any political map (or planar graph) using only 4 colors.

Any map or vertices on a planar graph can be colored using at most 4 colors. This is quite interesting story behind this. This is a first serious proof finished using automated theorem prover (Coq): https://en.wikipedia.org/wiki/Four_color_theorem.

An example where I use graph coloring: 22.7.1.

Let's try to color map of Europe:

```
#!/usr/bin/env python3
from z3 import *

borders={"Albania": ["Greece", "Kosovo", "Macedonia", "Montenegro"],
"Andorra": ["France", "Spain"],
"Austria": ["CzechRepublic", "Germany", "Hungary", "Italy", "Liechtenstein", "
   Slovakia", "Slovenia", "Switzerland"],
"Belarus": ["Latvia", "Lithuania", "Poland", "Ukraine"],
"Belgium": ["France", "Germany", "Luxembourg", "Netherlands"],
"BosniaHerzegovina": ["Croatia", "Montenegro", "Serbia"],
"Bulgaria": ["Greece", "Macedonia", "Romania", "Serbia"],
"Croatia": ["BosniaHerzegovina", "Hungary", "Montenegro", "Serbia", "Slovenia"],
"Cyprus": [],
"CzechRepublic": ["Austria", "Germany", "Poland", "Slovakia"],
"Denmark": ["Germany"],
"Estonia": ["Latvia"],
"Finland": ["Norway", "Sweden"],
"France": ["Andorra", "Belgium", "Germany", "Italy", "Luxembourg", "Monaco", "Spain",
    "Switzerland"],
"Germany": ["Austria", "Belgium", "CzechRepublic", "Denmark", "France", "Luxembourg",
    "Netherlands", "Poland", "Switzerland"],
"Greece": ["Albania", "Bulgaria", "Macedonia"],
"Hungary": ["Austria", "Croatia", "Romania", "Serbia", "Slovakia", "Slovenia", "
   Ukraine"],
"Iceland": [],
"Ireland": ["UnitedKingdom"],
"Italy": ["Austria", "France", "SanMarino", "Slovenia", "Switzerland", "VaticanCity"
   ],
"Kosovo": ["Albania", "Macedonia", "Montenegro", "Serbia"],
"Latvia": ["Belarus", "Estonia", "Lithuania"],
"Liechtenstein": ["Austria", "Switzerland"],
"Lithuania": ["Belarus", "Latvia", "Poland"],
"Luxembourg": ["Belgium", "France", "Germany"],
"Macedonia": ["Albania", "Bulgaria", "Greece", "Kosovo", "Serbia"],
"Malta": [],
"Moldova": ["Romania", "Ukraine"],
"Monaco": ["France"],
"Montenegro": ["Albania", "BosniaHerzegovina", "Croatia", "Kosovo", "Serbia"],
"Netherlands": ["Belgium", "Germany"],
"Norway": ["Finland", "Sweden"],
"Poland": ["Belarus", "CzechRepublic", "Germany", "Lithuania", "Slovakia", "Ukraine"
   ],
"Portugal": ["Spain"],
"Romania": ["Bulgaria", "Hungary", "Moldova", "Serbia", "Ukraine"],
"SanMarino": ["Italy"],
"Serbia": ["BosniaHerzegovina", "Bulgaria", "Croatia", "Hungary", "Kosovo", "
   Macedonia", "Montenegro", "Romania"],
"Slovakia": ["Austria", "CzechRepublic", "Hungary", "Poland", "Ukraine"],
"Slovenia": ["Austria", "Croatia", "Hungary", "Italy"],
"Spain": ["Andorra", "France", "Portugal"],
"Sweden": ["Finland", "Norway"],
"Switzerland": ["Austria", "France", "Germany", "Italy", "Liechtenstein"],
"Ukraine": ["Belarus", "Hungary", "Moldova", "Poland", "Romania", "Slovakia"],
"UnitedKingdom": ["Ireland"],
"VaticanCity": ["Italy"]}

s=Solver()

countries=list(borders.keys())
countries_total=len(countries)

country_color=[Int('country%d_color' % c) for c in range(countries_total)]

for i in range(countries_total):
    s.add(country_color[i]>=0)
    s.add(country_color[i]<4)

def country_name_to_idx(s):
    return countries.index(s)

for i in range(countries_total):
    for b in borders[countries[i]]:
        s.add(country_color[i] != country_color[country_name_to_idx(b)])

print (s.check())
m=s.model()

#for i in range(countries_total):
#    print (m[country_color[i]].as_long())

print ("coloring={")
for i in range(countries_total):
    color=m[country_color[i]].as_long()
    if color==0:
        s="1,0,0"
    elif color==1:
        s="0,1,0"
    elif color==2:
        s="0,0,1"
    elif color==3:
        s="1,1,0"
    print ("\tEntity[\"Country\", \""+countries[i]+"\"] -> RGBColor["+s+"], ")
print ("}")
```

The output is to be fed to Wolfram Mathematica – I'm using it to draw a map [1]:

[1] I copypasted pieces of it from https://www.wolfram.com/mathematica/new-in-10/entity-based-geocomputation/find-a-four-coloring-of-a-map-of-europe.html

```
coloring = {Entity["Country", "Lithuania"] -> RGBColor[0, 0, 1],
  Entity["Country", "Luxembourg"] -> RGBColor[0, 0, 1],
  Entity["Country", "Andorra"] -> RGBColor[0, 1, 0],
  Entity["Country", "Ireland"] -> RGBColor[1, 0, 0],
  Entity["Country", "Belarus"] -> RGBColor[0, 1, 0],
  Entity["Country", "Slovenia"] -> RGBColor[1, 0, 0],
  Entity["Country", "BosniaHerzegovina"] -> RGBColor[1, 1, 0],
  Entity["Country", "Belgium"] -> RGBColor[1, 1, 0],
  Entity["Country", "Spain"] -> RGBColor[0, 0, 1],
  Entity["Country", "Netherlands"] -> RGBColor[1, 0, 0],
  Entity["Country", "UnitedKingdom"] -> RGBColor[0, 1, 0],
  Entity["Country", "Denmark"] -> RGBColor[1, 0, 0],
  Entity["Country", "Poland"] -> RGBColor[1, 1, 0],
  Entity["Country", "Moldova"] -> RGBColor[0, 0, 1],
  Entity["Country", "Croatia"] -> RGBColor[0, 1, 0],
  Entity["Country", "Monaco"] -> RGBColor[0, 1, 0],
  Entity["Country", "Switzerland"] -> RGBColor[1, 1, 0],
  Entity["Country", "VaticanCity"] -> RGBColor[1, 0, 0],
  Entity["Country", "CzechRepublic"] -> RGBColor[1, 0, 0],
  Entity["Country", "Albania"] -> RGBColor[0, 0, 1],
  Entity["Country", "Estonia"] -> RGBColor[1, 0, 0],
  Entity["Country", "Kosovo"] -> RGBColor[1, 1, 0],
  Entity["Country", "Cyprus"] -> RGBColor[1, 0, 0],
  Entity["Country", "Italy"] -> RGBColor[0, 1, 0],
  Entity["Country", "Malta"] -> RGBColor[1, 0, 0],
  Entity["Country", "France"] -> RGBColor[1, 0, 0],
  Entity["Country", "Slovakia"] -> RGBColor[0, 1, 0],
  Entity["Country", "SanMarino"] -> RGBColor[0, 0, 1],
  Entity["Country", "Norway"] -> RGBColor[1, 0, 0],
  Entity["Country", "Iceland"] -> RGBColor[1, 0, 0],
  Entity["Country", "Montenegro"] -> RGBColor[1, 0, 0],
  Entity["Country", "Germany"] -> RGBColor[0, 1, 0],
  Entity["Country", "Ukraine"] -> RGBColor[1, 0, 0],
  Entity["Country", "Finland"] -> RGBColor[0, 1, 0],
  Entity["Country", "Macedonia"] -> RGBColor[0, 1, 0],
  Entity["Country", "Liechtenstein"] -> RGBColor[0, 1, 0],
  Entity["Country", "Latvia"] -> RGBColor[1, 1, 0],
  Entity["Country", "Bulgaria"] -> RGBColor[1, 0, 0],
  Entity["Country", "Romania"] -> RGBColor[0, 1, 0],
  Entity["Country", "Portugal"] -> RGBColor[1, 1, 0],
  Entity["Country", "Serbia"] -> RGBColor[0, 0, 1],
  Entity["Country", "Sweden"] -> RGBColor[0, 0, 1],
  Entity["Country", "Austria"] -> RGBColor[0, 0, 1],
  Entity["Country", "Greece"] -> RGBColor[1, 1, 0],
  Entity["Country", "Hungary"] -> RGBColor[1, 1, 0],}

GeoGraphics[{EdgeForm[
   Directive[Thin,
    Black]], {GeoStyling[#2], Tooltip[Polygon[#1], #1[[2]]]} & @@@
   coloring}]
```

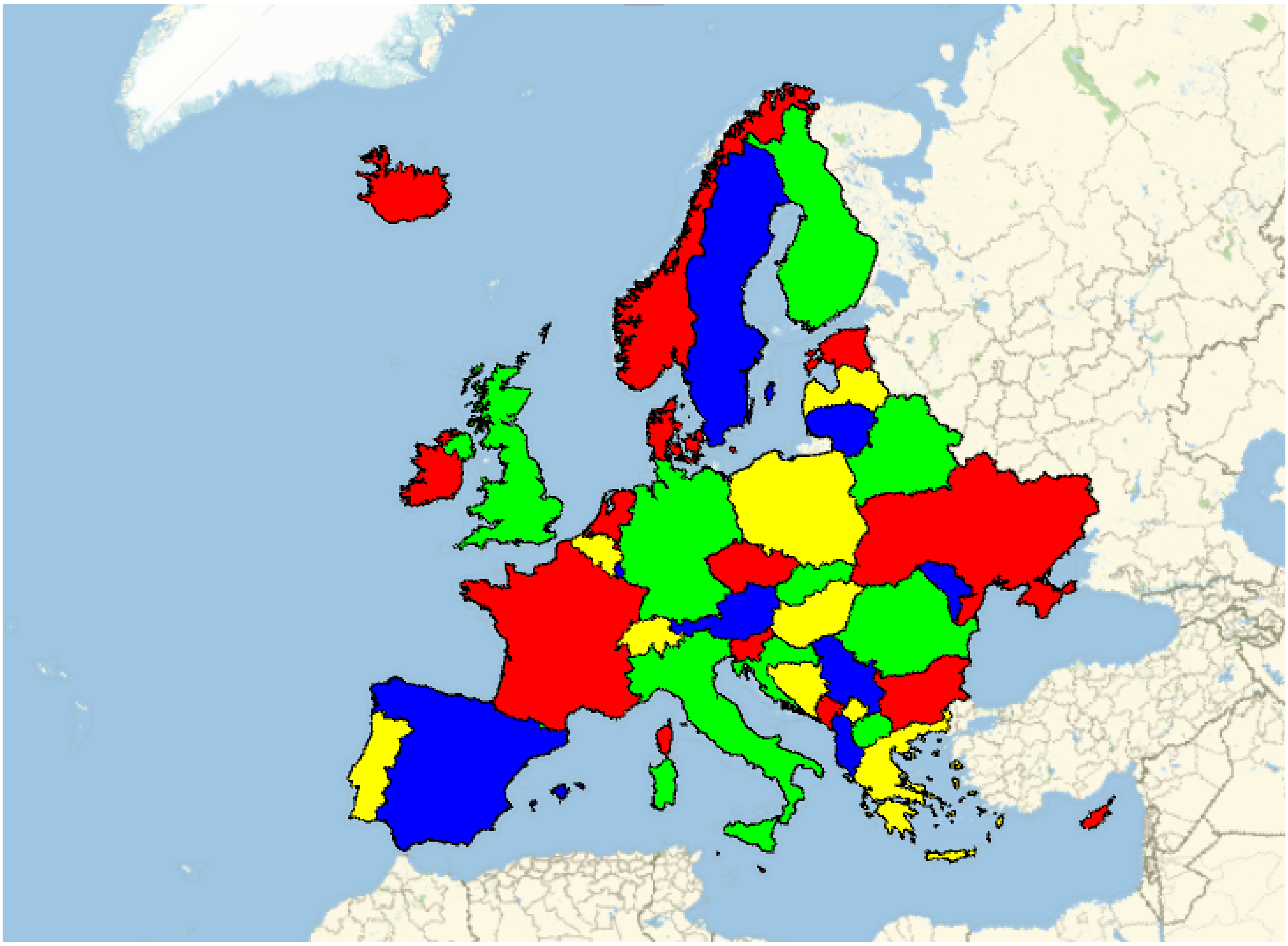


Figure 9.1: Colored map

9.1.1 MaxSMT or optimization problem

Now let's have fun. Out of pure whim, we may want to make as many countries colored as red as possible [2]:

```
s=Optimize()

...

s.maximize(Sum(*[If(country_color[i]==0, 1, 0) for i in range(countries_total)]))
```

[2] I took this idea from https://www.cs.cmu.edu/~bryant/boolean/macgregor.html

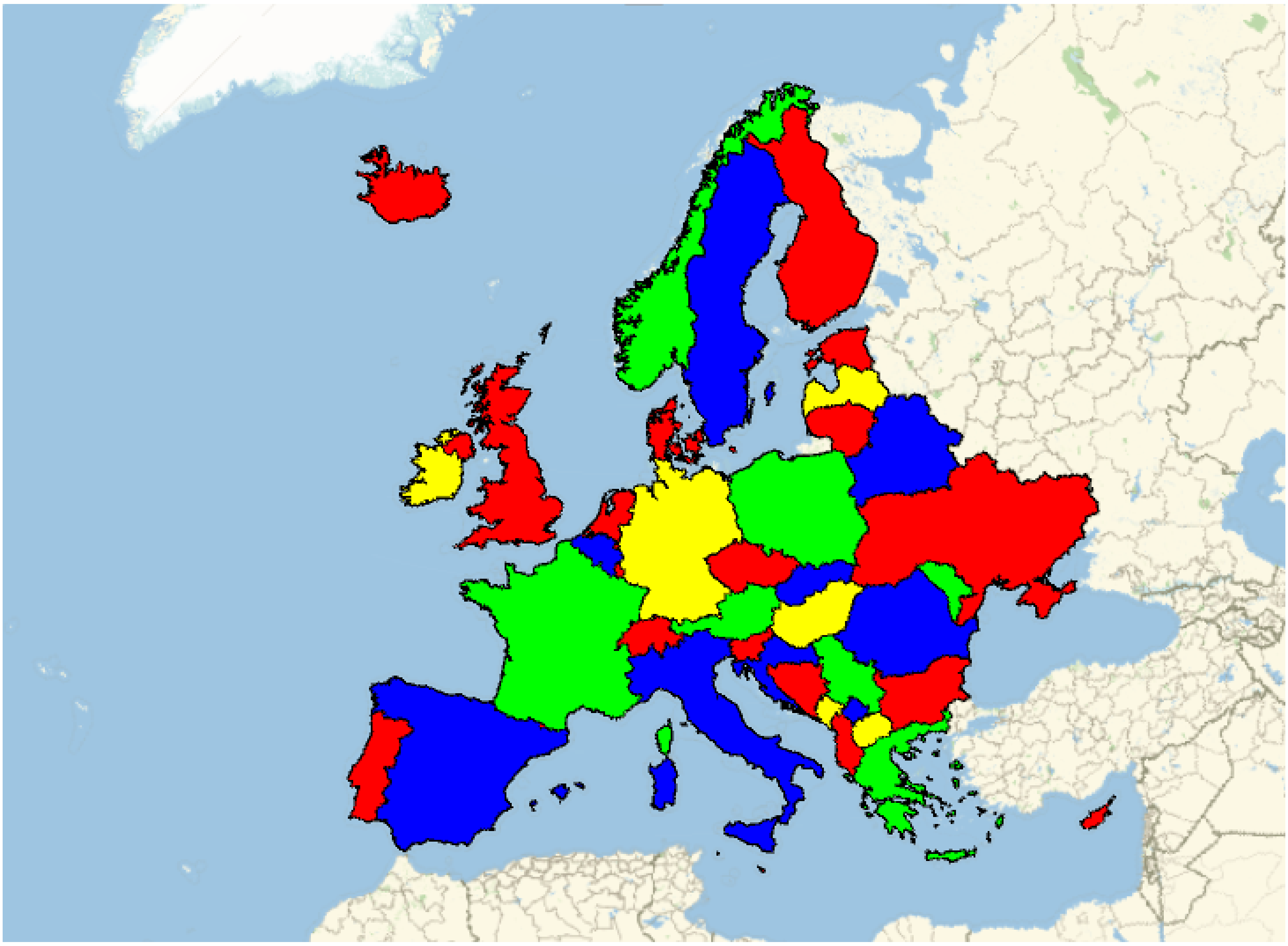


Figure 9.2: The map

Listing 9.1: Statistics

```
     22 red
      7 green
     10 blue
      6 yellow
```

Or maybe we have a shortage of red paint?

```
s.minimize(Sum(*[If(country_color[i]==0, 1, 0) for i in range(countries_total)]))
```

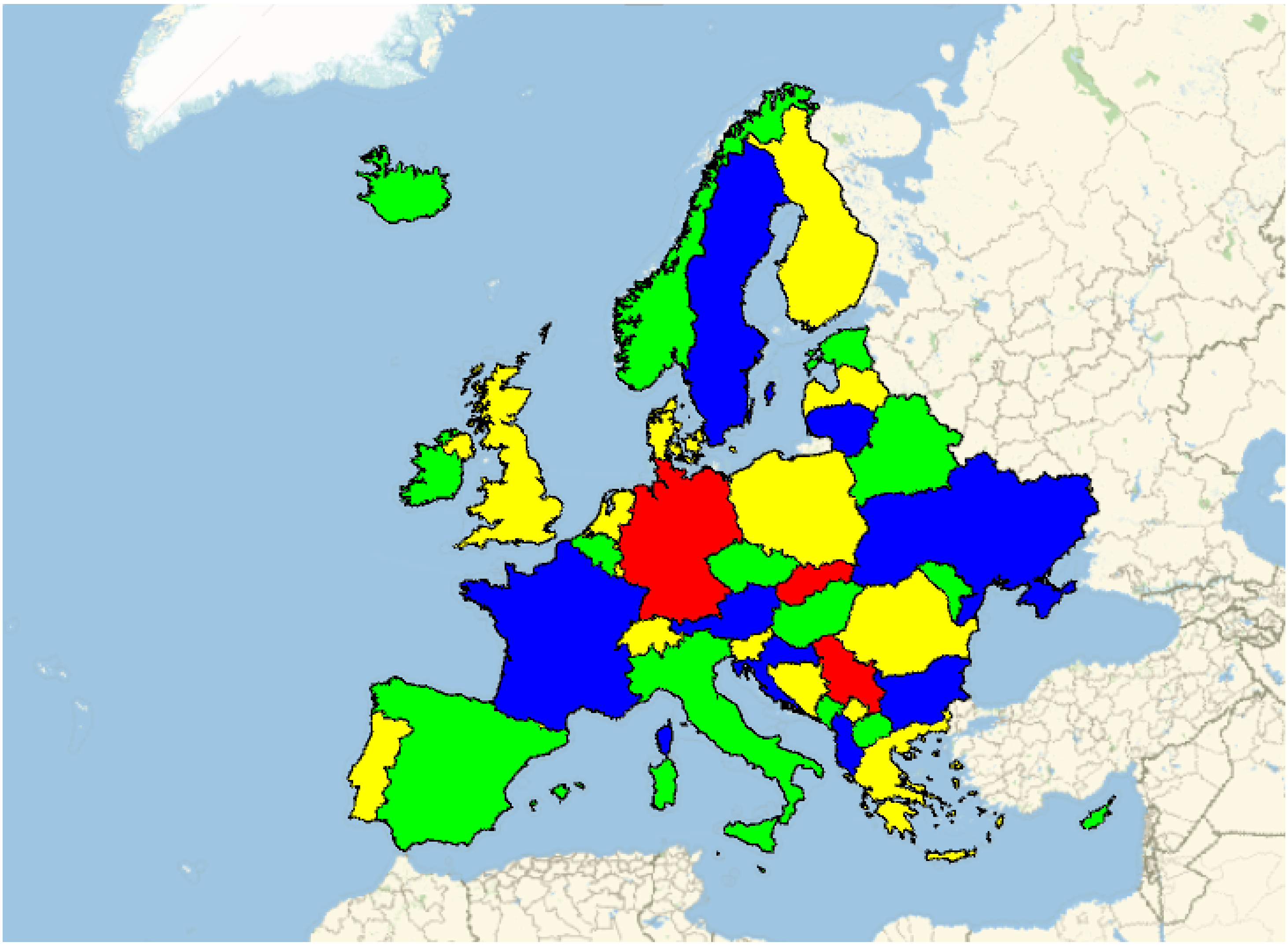


Figure 9.3: The map

Listing 9.2: Statistics

```
      3 red
     17 green
     10 blue
     15 yellow
```

Two colors can be minimized/maximized simultaneously:

```
...
s.minimize(Sum(*[If(Or(country_color[i]==0, country_color[i]==1), 1, 0) for i in
    range(countries_total)]))
...
```

9.2 Using graph coloring in scheduling

I've found this problem in the "Discrete Structures, Logic and Computability" book by James L. Hein:

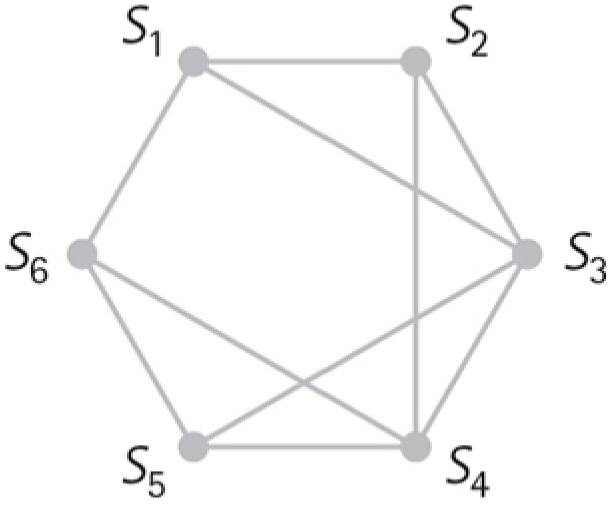


Figure 9.4: Graph

Suppose some people form committees to do various tasks. The problem is to schedule the committee meetings in as few time slots as possible. To simplify the discussion, we'll represent each person with a number. For example, let S = 1, 2, 3, 4, 5, 6, 7 represent a set of seven people, and suppose they have formed six three-person committees as follows:

S1 = 1, 2, 3, S2 = 2, 3, 4, S3 = 3, 4, 5, S4 = 4, 5, 6, S5 = 5, 6, 7, S6 = 1, 6, 7.

We can model the problem with the graph pictured in Figure 1.4.4, where the committee names are the vertices and an edge connects two vertices if a person belongs to both committees represented by the vertices. If each committee meets for one hour, what is the smallest number of hours needed for the committees to do their work? From the graph, it follows that an edge between two committees means that they have at least one member in common. Thus, they cannot meet at the same time. No edge between committees means that they can meet at the same time. For example, committees S1 and S4 can meet the first hour. Then committees S2 and S5 can meet the second hour. Finally, committees S3 and S6 can meet the third hour. Can you see why three hours is the smallest number of hours that the six committees can meet?

And this is solution:

```
#!/usr/bin/env python3

import itertools
from z3 import *

# 7 peoples, 6 committees

S={}
S[1]=set([1, 2, 3])
S[2]=set([2, 3, 4])
S[3]=set([3, 4, 5])
S[4]=set([4, 5, 6])
S[5]=set([5, 6, 7])
S[6]=set([1, 6, 7])

committees=len(S)

s=Solver()

Color_or_Hour=[Int('Color_or_hour_%d' % i) for i in range(committees)]

# enumerate all possible pairs of committees:
```

```
for pair in itertools.combinations(S, r=2):
    # if intersection of two sets has *something* (i.e., if two committees has at least one person in common):
    if len(S[pair[0]] & S[pair[1]])>0:
        # ... add an edge between vertices (or persons) -- these colors (or hours) must differ:
        s.add(Color_or_Hour[pair[0]-1] != Color_or_Hour[pair[1]-1])

# limit all colors (or hours) in 0..2 range (3 colors/hours):
for i in range(committees):
    s.add(And(Color_or_Hour[i]>=0, Color_or_Hour[i]<=2))

assert s.check()==sat
m=s.model()
#print (m)

schedule={}

for i in range(committees):
    hour=m[Color_or_Hour[i]].as_long()
    if hour not in schedule:
        schedule[hour]=[]
    schedule[hour].append(i+1)

for t in schedule:
    print ("hour:", t, "committees:", schedule[t])
```

The result:

```
hour: 0 committees: [1, 4]
hour: 1 committees: [2, 5]
hour: 2 committees: [3, 6]
```

If you increase total number of hours to 4, the result is somewhat sparser:

```
hour: 0 committees: [3]
hour: 1 committees: [1, 4]
hour: 2 committees: [2, 5]
hour: 3 committees: [6]
```

9.3 Another example

In place of epigraph:

> Suppose that you are organizing housing accommodations for a group of four hundred university students. Space is limited and only one hundred of the students will receive places in the dormitory. To complicate matters, the Dean has provided you with a list of pairs of incompatible students, and requested that no pair from this list appear in your final choice.

(https://www.claymath.org/millennium-problems/p-vs-np-problem)

What if we want to divide our community/company/university by groups. There are 16 persons and, which must be divided by 4 groups, 4 persons in each. However, several persons hate each other, maybe, for personal reasons. Can we group all them so the ”haters” would be separated?

```
#!/usr/bin/env python3

```

```
from z3 import *

# 16 peoples, 4 groups

PERSONS=16
GROUPS=4

s=Solver()

person=[Int('person_%d' % i) for i in range(PERSONS)]

# each person must belong to some group in 0..GROUPS range:
for i in range(PERSONS):
    s.add(And(person[i]>=0, person[i]<GROUPS))

# each pair of persons can't be in the same group, because they hate each other.
# IOW, we add an edge between vertices.
s.add(person[0] != person[7])
s.add(person[0] != person[8])
s.add(person[0] != person[9])
s.add(person[2] != person[9])
s.add(person[9] != person[14])
s.add(person[11] != person[15])
s.add(person[11] != person[1])
s.add(person[11] != person[2])
s.add(person[11] != person[9])
s.add(person[10] != person[1])

persons_in_group=[Int('persons_in_group_%d' % i) for i in range(GROUPS)]

def count_persons_in_group(g):
    """
    Form expression like:

    If(person_0 == g, 1, 0) +
    If(person_1 == g, 1, 0) +
    If(person_2 == g, 1, 0) +
    ...
    If(person_15 == g, 1, 0)

    """
    return Sum(*[If(person[i]==g, 1, 0) for i in range(PERSONS)])

# each group must have 4 persons:
for g in range(GROUPS):
    s.add(count_persons_in_group(g)==4)

assert s.check()==sat
m=s.model()

groups={}
for i in range(PERSONS):
    g=m[person[i]].as_long()
    if g not in groups:
        groups[g]=[]
    groups[g].append(i)

for g in groups:
    print ("group %d, persons:" % g, groups[g])
```

The result:

```
group 0, persons: [1, 2, 5, 8]
group 1, persons: [4, 7, 9, 12]
group 2, persons: [0, 3, 11, 13]
group 3, persons: [6, 10, 14, 15]
```

9.4 Register allocation using graph coloring

This is an implementation of nuth-Morris-Pratt algorithm, it searches for a substring in a string [3].

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char *kmp_search(char *haystack, size_t haystack_size, char *needle, size_t
   needle_size);

int64_t T[1024];

char *kmp_search(char *haystack, size_t haystack_size, char *needle, size_t
   needle_size)
{
        //int *T;
        int64_t i, j;
        char *result = NULL;

        if (needle_size==0)
                return haystack;

        /* Construct the lookup table */
        //T = (int*) malloc((needle_size+1) * sizeof(int));
        T[0] = -1;
        for (i=0; i<needle_size; i++)
        {
                T[i+1] = T[i] + 1;
                while (T[i+1] > 0 && needle[i] != needle[T[i+1]-1])
                        T[i+1] = T[T[i+1]-1] + 1;
        }

        /* Perform the search */
        for (i=j=0; i<haystack_size; )
        {
                if (j < 0 || haystack[i] == needle[j])
                {
                        ++i, ++j;
                        if (j == needle_size)
                        {
                                result = haystack+i-j;
                                break;
                        }
                }
                else j = T[j];
        }

        //free(T);
        return result;
}

char* helper(char* haystack, char* needle)
{
```

[3] Copypasted from http://cprogramming.com/snippets/source-code/knuthmorrispratt-kmp-string-search-algorithm

```
        return kmp_search(haystack, strlen(haystack), needle, strlen(needle));
};

int main()
{
        printf ("%s\n", helper("hello world", "world"));
        printf ("%s\n", helper("hello world", "ell"));
};
```

... as you can see, I simplified it a bit, there are no more calls to malloc/free and T[] array is now global.

Then I compiled it using GCC 7.3 x64 and reworked assembly listing a little, now there are no registers, but rather vXX variables, each one is assigned only once, in a SSA manner. No variable assigned more than once. This is AT&T syntax.

```
#RDI - haystack       v1
#RSI - haystack_size  v2
#RDX - needle         v3
#RCX - needle_size    v4

        .text
        .globl  kmp_search
        .type   kmp_search, @function
kmp_search:
                                        # v1  v2  v3  v4  v5  v6  v7  v8  v9  v10 v11 v12 v13 v14 v15 v16
        testq   %v4, %v4                #  |   |   |   U
        movq    %v1, %rax               #  U   |   |   |
        je      .exit                   #  |   |   |   |
        leaq    (%v4,%v3), %v7          #  |   |   U   U           D
        movq    %v3, %v5                #  |   |   U   |   D       |
        leaq    8+T(%rip), %v9          #  |   |   |   |   |       |       D
        movq    $-1, T(%rip)            #  |   |   |   |   |       |       |
        cmpq    %v5, %v7                #  |   |   |   |   U       U       |
        leaq    -8(%v9), %v8            #  |   |   |   |   |       |   D   U
        je      .L20                    #  |   |   |   |   |       |   |   |
.L6:                                    #  |   |   |   |   |       |   |   |
        movq    -8(%v9), %v14           #  |   |   |   |   |       |   |   U                    D
        addq    $1, %v14                #  |   |   |   |   |       |   |   |                    U
        testq   %v14, %v14              #  |   |   |   |   |       |   |   |                    U
        movq    %v14, (%v9)             #  |   |   |   |   |       |   |   U                    U
        jle     .L4                     #  |   |   |   |   |       |   |   |                    |
        movzbl  (%v5), %v11             #  |   |   |   |   U       |   |   |        D           |
        cmpb    %v11_byte, -1(%v3,%v14) #  |   |   U   |   |       |   |   |        U           U
        jne     .L5                     #  |   |   |   |   |       |   |   |                    |
        jmp     .L4                     #  |   |   |   |   |       |   |   |                    |
.L21:                                   #  |   |   |   |   |       |   |   |                    |
        movzbl  -1(%v3,%v14), %v12      #  |   |   U   |   |       |   |   |            D       U
        cmpb    %v12_byte, (%v5)        #  |   |   |   |   U       |   |   |            U       |
        je      .L4                     #  |   |   |   |   |       |   |   |                    |
.L5:                                    #  |   |   |   |   |       |   |   |                    |
        movq    -8(%v8,%v14,8), %v15    #  |   |   |   |   |       |   U   |                    U   D
        addq    $1, %v15                #  |   |   |   |   |       |       |                        U
        testq   %v15, %v15              #  |   |   |   |   |       |       |                        U
        movq    %v15, (%v9)             #  |   |   |   |   |       |       U                        U
        jg      .L21                    #  |   |   |   |   |       |       |
.L4:                                    #  |   |   |   |   |       |       |
        addq    $1, %v5                 #  |   |   |   |   U       |       |
        addq    $8, %v9                 #  |   |   |   |   |       |       U
        cmpq    %v5, %v7                #  |   |   |   |   U       U       |
        jne     .L6                     #  |   |   |   |   |       |       |
.L20:
                                        # v1  v2  v3  v4  v5  v6  v7  v8  v9  v10 v11 v12 v13 v14 v15 v16
                                        #  |   |   |   |
        leaq    T(%rip), %v6            #  |   |   |   |       D
        xorq    %v16, %v16              #  |   |   |   |       |                                        D
        xorq    %v10, %v10              #  |   |   |   |       |                D                       |
.L7:                                    #  |   |   |   |       |                |                       |
        cmpq    %v10, %v2               #  |   U   |   |       |                U                       |
        jbe     .L22                    #  |   |   |   |       |                |                       |
.L11:                                   #  |   |   |   |       |                |                       |
        testq   %v16, %v16              #  |   |   |   |       |                |                       U
        js      .L8                     #  |   |   |   |       |                |                       |
        movzbl  (%v3,%v16), %v13        #  |   |   U   |       |                |           D           U
        cmpb    %v13_byte, (%v1,%v10)   #  U   |       |       |                U           U           |
        je      .L8                     #  |   |       |       |                |                       |
        cmpq    %v10, %v2               #  |   U       |       |                U                       |
        movq    (%v6,%v16,8), %v16      #  |           |       U                |                       U
        ja      .L11                    #  |           |                        |                       |
.L22:                                   #  |           |                        |                       |
        xorq    %rax, %rax              #  |           |                        |                       |
        ret                             #  |           |                        |                       |
.L8:                                    #  |           |                        |                       |
        addq    $1, %v16                #  |           |                        |                       U
        addq    $1, %v10                #  |           |                        U                       |
        cmpq    %v16, %v4               #  |           U                        |                       U
        jne     .L7                     #  |           |                        |
        subq    %v4, %v10               #  |           U                        U
        leaq    (%v1,%v10), %rax        #  U                                    U
.exit:
        ret
        .comm   T,8192,32
```

Dangling "noodles" you see at right are "live ranges" of each vXX variable. "D" means "defined", "U" - "used" or "used and then defined again". Whenever live range is started, we need to allocate variable (in a register or a local stack). When it's ending, we do not need to keep it somewhere in storage (in a register or a local stack).

As you can see, the function has two parts: preparation and processing. You can clearly see how live ranges are divided by two groups, except of first 4 variables, which are function arguments.

You see, there are 16 variables. But we want to use as small number of registers, as possible. If several live ranges are present at some address or point of time, these variables cannot be allocated in the same register.

This is how we can assign a register to each live range using Z3 SMT-solver:

```
#!/usr/bin/env python3
from z3 import *

def attempt(colors):

    v=[Int('v%d' % i) for i in range(18)]

    s=Solver()

    for i in range(18):
        s.add(And(v[i]>=0, v[i]<colors))

    # a bit redundant, but that is not an issue:
    s.add(Distinct(v[1], v[2], v[3], v[4]))
    s.add(Distinct(v[1], v[2], v[3], v[4], v[7]))
    s.add(Distinct(v[1], v[2], v[3], v[4], v[5], v[7]))
    s.add(Distinct(v[1], v[2], v[3], v[4], v[5], v[7], v[9]))
    s.add(Distinct(v[1], v[2], v[3], v[4], v[5], v[7], v[8], v[9]))
    s.add(Distinct(v[1], v[2], v[3], v[4], v[5], v[7], v[8], v[9], v[14]))
    s.add(Distinct(v[1], v[2], v[3], v[4], v[5], v[7], v[8], v[9], v[11], v[14]))
    s.add(Distinct(v[1], v[2], v[3], v[4], v[5], v[7], v[8], v[9], v[12], v[14]))
    s.add(Distinct(v[1], v[2], v[3], v[4], v[5], v[7], v[8], v[9], v[14], v[15]))
    s.add(Distinct(v[1], v[2], v[3], v[4], v[5], v[7], v[9], v[14], v[15]))
    s.add(Distinct(v[1], v[2], v[3], v[4], v[5], v[7], v[9], v[14]))
    s.add(Distinct(v[1], v[2], v[3], v[4], v[5], v[7], v[14]))
    s.add(Distinct(v[1], v[2], v[3], v[4]))
    s.add(Distinct(v[1], v[2], v[3], v[4], v[6]))
    s.add(Distinct(v[1], v[2], v[3], v[4], v[6], v[16]))
    s.add(Distinct(v[1], v[2], v[3], v[4], v[6], v[10], v[16]))
    s.add(Distinct(v[1], v[2], v[3], v[4], v[6], v[10], v[13], v[16]))
    s.add(Distinct(v[1], v[2], v[4], v[6], v[10], v[13], v[16]))
    s.add(Distinct(v[1], v[2], v[4], v[6], v[10], v[16]))
    s.add(Distinct(v[1], v[4], v[6], v[10], v[16]))
    s.add(Distinct(v[1], v[4], v[10], v[16]))
    s.add(Distinct(v[1], v[4], v[10]))
    s.add(Distinct(v[1], v[10]))

    registers=["RDI", "RSI", "RDX", "RCX", "R8", "R9", "R10", "R11", "R12", "R13", "
       R14", "R15"]
    # first 4 variables are function arguments and they are always linked to rdi/rsi/rdx/rcx:
    s.add(v[1]==0)
    s.add(v[2]==1)
    s.add(v[3]==2)
    s.add(v[4]==3)

    if s.check()==sat:
        print ("* colors=", colors)
        m=s.model()
        for i in range(1, 17):
            print ("v%d=%s" % (i, registers[m[v[i]].as_long()]))

for colors in range(12, 0, -1):
    attempt(colors)
```

What we've got for 12, 11 and 10 registers:

```
* colors= 12
v1=RDI
v2=RSI
v3=RDX
v4=RCX
v5=R8
v6=R8
v7=R9
v8=R10
v9=R11
v10=R9
v11=R12
v12=R12
v13=R10
v14=R13
v15=R14
v16=R11
* colors= 11
v1=RDI
v2=RSI
v3=RDX
v4=RCX
v5=R14
v6=R8
v7=R13
v8=R8
v9=R12
v10=R11
v11=R9
v12=R11
v13=R9
v14=R10
v15=R9
v16=R10
* colors= 10
v1=RDI
v2=RSI
v3=RDX
v4=RCX
v5=R10
v6=R11
v7=R8
v8=R13
v9=R11
```

```
v10=R10
v11=R12
v12=R12
v13=R8
v14=R9
v15=R12
v16=R9
```

It's not possible to assign 9 or less registers. 10 is a minimum.

Now all I do is replacing vXX variables to registers the SMT-solver offered:

```
#RDI - haystack       v1
#RSI - haystack_size  v2
#RDX - needle         v3
#RCX - needle_size    v4

        .text
        .globl  kmp_search
        .type   kmp_search, @function
kmp_search:
                                        # v1  v2  v3  v4  v5  v6  v7  v8  v9  v10 v11 v12 v13 v14 v15 v16
        testq   %rcx, %rcx              #  |   |   |   U
        movq    %rdi, %rax              #  U   |   |   |
        je      .exit                   #  |   |   |   |
        leaq    (%rcx,%rdx), %r8        #  |   |   U   U           D
        movq    %rdx, %r10              #  |   |   U   |   D       |
        leaq    8+T(%rip), %r11         #  |   |   |   |   |       |       D
        movq    $-1, T(%rip)            #  |   |   |   |   |       |       |
        cmpq    %r10, %r8               #  |   |   |   |   U       U       |
        leaq    -8(%r11), %r13          #  |   |   |   |   |       |   D   U
        je      .L20                    #  |   |   |   |   |       |   |   |
.L6:                                    #  |   |   |   |   |       |   |   |
        movq    -8(%r11), %r9           #  |   |   |   |   |       |   |   U                    D
        addq    $1, %r9                 #  |   |   |   |   |       |   |   |                    U
        testq   %r9, %r9                #  |   |   |   |   |       |   |   |                    U
        movq    %r9, (%r11)             #  |   |   |   |   |       |   |   U                    U
        jle     .L4                     #  |   |   |   |   |       |   |   |                    |
        movzbl  (%r10), %r12d           #  |   |   |   |   U       |   |   |        D           |
        cmpb    %r12b, -1(%rdx,%r9)     #  |   |   U   |   |       |   |   |        U           U
        jne     .L5                     #  |   |   |   |   |       |   |   |                    |
        jmp     .L4                     #  |   |   |   |   |       |   |   |                    |
.L21:                                   #  |   |   |   |   |       |   |   |                    |
        movzbl  -1(%rdx,%r9), %r12d     #  |   |   U   |   |       |   |   |            D       U
        cmpb    %r12b, (%r10)           #  |   |   |   |   U       |   |   |            U       |
        je      .L4                     #  |   |   |   |   |       |   |   |                    |
.L5:                                    #  |   |   |   |   |       |   |   |                    |
        movq    -8(%r13,%r9,8), %r12    #  |   |   |   |   |       |   U   |                    U   D
        addq    $1, %r12                #  |   |   |   |   |       |       |                        U
        testq   %r12, %r12              #  |   |   |   |   |       |       |                        U
        movq    %r12, (%r11)            #  |   |   |   |   |       |       U                        U
        jg      .L21                    #  |   |   |   |   |       |       |
.L4:                                    #  |   |   |   |   |       |       |
        addq    $1, %r10                #  |   |   |   |   U       |       |
        addq    $8, %r11                #  |   |   |   |   |       |       U
        cmpq    %r10, %r8               #  |   |   |   |   U       U       |
        jne     .L6                     #  |   |   |   |   |       |       |
.L20:
                                        # v1  v2  v3  v4  v5  v6  v7  v8  v9  v10 v11 v12 v13 v14 v15 v16
                                        #  |   |   |   |
        leaq    T(%rip), %r11           #  |   |   |   |       D
        xorq    %r9, %r9                #  |   |   |   |       |                                        D
        xorq    %r10, %r10              #  |   |   |   |       |                D                       |
.L7:                                    #  |   |   |   |       |                |                       |
        cmpq    %r10, %rsi              #  |   U   |   |       |                U                       |
        jbe     .L22                    #  |   |   |   |       |                |                       |
.L11:                                   #  |   |   |   |       |                |                       |
        testq   %r9, %r9                #  |   |   |   |       |                |                       U
        js      .L8                     #  |   |   |   |       |                |                       |
        movzbl  (%rdx,%r9), %r8d       #  |   |   U   |       |                |           D           U
        cmpb    %r8b, (%rdi,%r10)       #  U   |       |       |                U           U           |
        je      .L8                     #  |   |       |       |                |                       |
        cmpq    %r10, %rsi              #  |   U       |       |                U                       |
        movq    (%r11,%r9,8), %r9       #  |           |       U                |                       U
        ja      .L11                    #  |           |                        |                       |
.L22:                                   #  |           |                        |                       |
        xorq    %rax, %rax              #  |           |                        |                       |
        ret                             #  |           |                        |                       |
.L8:                                    #  |           |                        |                       |
        addq    $1, %r9                 #  |           |                        |                       U
        addq    $1, %r10                #  |           |                        U                       |
        cmpq    %r9, %rcx               #  |           U                        |                       U
        jne     .L7                     #  |           |                        |
        subq    %rcx, %r10              #  |           U                        U
        leaq    (%rdi,%r10), %rax       #  U                                    U
.exit:
        ret
        .comm   T,8192,32
```

That works and it's almost the same as GCC does.

The problem of register allocation as a graph coloring problem: each live range is a vertex. It a live range must coexist with another live range, put an edge between vertices, that means, vertices cannot share same color. Color reflecting register number.

Almost all compilers (except simplest) do this in code generator. They use simpler algorithms, though, instead of such a heavy machinery as SAT/SMT solvers.

See also: https://en.wikipedia.org/wiki/Register_allocation, https://en.wikipedia.org/wiki/Live_variable_analysis.

Since graph coloring can have many solutions, you can probably hide some information in ”coloring”. [See about ”Lehmer code” in Untitled collection of math notes[4]].

[4] https://xn--mxacd.xn--qxam/math-notes.pdf

Chapter 10

Knapsack problems

10.1 Popsicles

Found this problem at http://artofproblemsolving.com/wiki/index.php?title=2017_AMC_12A_Problems/Problem_1:

> Pablo buys popsicles for his friends. The store sells single popsicles for \$1 each, 3-popsicle boxes for \$2, and 5-popsicle boxes for \$3. What is the greatest number of popsicles that Pablo can buy with \$8?

This is optimization problem, and the solution using z3:

```
#!/usr/bin/env python3
from z3 import *

box1pop, box3pop, box5pop = Ints('box1pop box3pop box5pop')
pop_total = Int('pop_total')
cost_total = Int('cost_total')

s=Optimize()

s.add(pop_total == box1pop*1 + box3pop*3 + box5pop*5)
s.add(cost_total == box1pop*1 + box3pop*2 + box5pop*3)

s.add(cost_total==8)

s.add(box1pop>=0)
s.add(box3pop>=0)
s.add(box5pop>=0)

s.maximize(pop_total)

print (s.check())
print (s.model())
```

The result:

```
sat
[box3pop = 1,
 box5pop = 2,
 cost_total = 8,
 pop_total = 13,
 box1pop = 0]
```

10.1.1 SMT-LIB 2.x

```
; tested with MK85 and Z3

(declare-fun box1pop () (_ BitVec 16))
(declare-fun box3pop () (_ BitVec 16))
(declare-fun box5pop () (_ BitVec 16))
(declare-fun pop_total () (_ BitVec 16))
(declare-fun cost_total () (_ BitVec 16))

(assert (=
        ((_ zero_extend 16) pop_total)
        (bvadd
                ((_ zero_extend 16) box1pop)
                (bvmul ((_ zero_extend 16) box3pop) #x00000003)
                (bvmul ((_ zero_extend 16) box5pop) #x00000005)
        )))

(assert (=
        ((_ zero_extend 16) cost_total)
        (bvadd
                ((_ zero_extend 16) box1pop)
                (bvmul ((_ zero_extend 16) box3pop) #x00000002)
                (bvmul ((_ zero_extend 16) box5pop) #x00000003)
        )))

(assert (= cost_total #x0008))

(maximize pop_total)

(check-sat)
(get-model)

; correct solution:

;(model
;        (define-fun box1pop () (_ BitVec 16) (_ bv0 16)) ; 0x0
;        (define-fun box3pop () (_ BitVec 16) (_ bv1 16)) ; 0x1
;        (define-fun box5pop () (_ BitVec 16) (_ bv2 16)) ; 0x2
;        (define-fun pop_total () (_ BitVec 16) (_ bv13 16)) ; 0xd
;        (define-fun cost_total () (_ BitVec 16) (_ bv8 16)) ; 0x8
;)
```

10.2 Organize your backups

In the era of audio cassettes (1980s, 1990s), many music lovers recorded their own mix tapes with tracks/songs they like. Each side of audio cassette was 30 or 45 minutes. The problem was to make such an order of songs, so that a minimal irritating ”silent” track at the end of each side would left. Surely, you wanted to use the space as efficiently, as possible.

This is classic *bin packing problem*: all bins (or cassettes) are equally sized.

Now let’s advance this problem further: bins can be different. You may want to backup your files to all sorts of storages you have: DVD-RWs, USB-sticks, remote hosts, cloud storage accounts, etc. This is a *Multiple Knapsack Problem* – you’ve got several knapsacks with different sizes.

```
#!/usr/bin/env python3
from z3 import *
import itertools

storages=[700, 30, 100, 800, 100, 800, 300, 150, 60, 500, 1000]
files=[18, 57, 291, 184, 167, 496, 45, 368, 144, 428, 15, 100, 999]
```

```
files_n=len(files)
files_t=sum(files)

print ("total storage we need", files_t)

def try_to_fit_into_storages(storages_to_be_used):
    t=len(storages_to_be_used)
    # for each server:
    storage_occupied=[Int('storage%d_occupied' % i) for i in range(t)]
    # which storage the file occupies?
    file_in_storage=[Int('file%d_in_storage' % i) for i in range(files_n)]

    # how much storage we have in picked storages, total?
    storage_t=0
    for i in range(t):
        storage_t=storage_t+storages[storages_to_be_used[i]]
    # skip if the sum of storage in picked storages is too small:
    if files_t > storage_t:
        return

    print ("trying to fit all the files into storages:", storages_to_be_used,end=' ')

    s=Solver()

    # all files must occupy some storage:
    for i in range(files_n):
        s.add(And(file_in_storage[i]>=0, file_in_storage[i]<t))

    for i in range(t):
        """
        here we generate expression like:

        If(file1_in_storage == 4, 57, 0) +
        If(file1_in_storage == 4, 291, 0) +
        If(file1_in_storage == 4, 184, 0) +
        If(file1_in_storage == 4, 167, 0) +
        If(file1_in_storage == 4, 496, 0) +
        If(file1_in_storage == 4, 45, 0) +
        If(file1_in_storage == 4, 368, 0) +
        If(file1_in_storage == 4, 144, 0) +
        If(file1_in_storage == 4, 428, 0) +
        If(file1_in_storage == 4, 15, 0)

        ... in plain English - if a file is in storage, add its size to the final sum

        """

        s.add(storage_occupied[i]==Sum([If(file_in_storage[f]==i, files[f], 0) for f
            in range(files_n)]))

        # ... but sum of all files in each storage must be lower than what we have in
             the storage:
        s.add(And(storage_occupied[i]>=0, storage_occupied[i]<=storages[
            storages_to_be_used[i]]))
    if s.check()==sat:
        print ("sat")
        print ("* solution (%d storages):" % t)
        m=s.model()
        #print (m)
        for i in range(t):
            print ("storage%d (total=%d):" % (storages_to_be_used[i], storages[
                storages_to_be_used[i]]))
            for f in range(files_n):
                if m[file_in_storage[f]].as_long()==i:
                    print ("    file%d (%d)" % (f, files[f]))
            print ("allocated on storage=%d" % (m[storage_occupied[i]].as_long()),end
                =' ')
            print ("free on storage=%d" % (storages[storages_to_be_used[i]] - m[
                storage_occupied[i]].as_long()))
        print ("total in all storages=%d" % storage_t)
        print ("allocated on all storages=%d%%" % ((float(files_t)/float(storage_t))
            *100))
        print ("")
        return True
    else:
        print ("unsat")
        return False

# how many storages we need? start with 2:
found_solution=False
for storages_to_pick in range(2, len(storages)+1):

    # we use Python itertools to find all combinations
    # in other words, pick $storages_to_pick$ storages from all storages, and
        enumerate all possible ways to choose from them.
    # see also: https://en.wikipedia.org/wiki/Combination
    for storages_to_be_used in itertools.combinations(range(len(storages)), r=
        storages_to_pick):
        # for some reasons, we may want to always use storage0
        # skip all sets, where no storage0 present:
        if 0 not in storages_to_be_used:
            continue
        if try_to_fit_into_storages(storages_to_be_used):
            found_solution=True
    # after we've got some solutions for $storages_to_pick$, stop:
    if found_solution:
        break
```

(https://smt.st/current_tree/knapsack/backup/backup.py)

Choose any solution you like:

```
trying to fit all the files into storages: (0, 3, 5, 7, 10) sat
* solution (5 storages):
storage0 (total=700):
    file0 (18)
    file3 (184)
    file9 (428)
allocated on storage=630 free on storage=70
storage3 (total=800):
    file2 (291)
    file5 (496)
allocated on storage=787 free on storage=13
storage5 (total=800):
    file1 (57)
    file4 (167)
    file6 (45)
    file7 (368)
    file8 (144)
allocated on storage=781 free on storage=19
storage7 (total=150):
    file10 (15)
    file11 (100)
```

```
allocated on storage=115 free on storage=35
storage10 (total=1000):
    file12 (999)
allocated on storage=999 free on storage=1
total in all storages=3450
allocated on all storages=96%

trying to fit all the files into storages: (0, 3, 5, 8, 10) sat
* solution (5 storages):
storage0 (total=700):
    file3 (184)
    file5 (496)
allocated on storage=680 free on storage=20
storage3 (total=800):
    file1 (57)
    file4 (167)
    file8 (144)
    file9 (428)
allocated on storage=796 free on storage=4
storage5 (total=800):
    file0 (18)
    file2 (291)
    file7 (368)
    file11 (100)
allocated on storage=777 free on storage=23
storage8 (total=60):
    file6 (45)
    file10 (15)
allocated on storage=60 free on storage=0
storage10 (total=1000):
    file12 (999)
allocated on storage=999 free on storage=1
total in all storages=3360
allocated on all storages=98%
```

Now something practical. You may want to store each file twice. And no pair must reside on a single storage. Not a problem, just make two arrays of variables:

```
...

     file1_in_storage=[Int('file1_%d_in_storage' % i) for i in range(files_n)]
     file2_in_storage=[Int('file2_%d_in_storage' % i) for i in range(files_n)]

...

         s.add(And(file1_in_storage[i]>=0, file1_in_storage[i]<t))
         s.add(And(file2_in_storage[i]>=0, file2_in_storage[i]<t))
         # no pair can reside on one storage:
         s.add(file1_in_storage[i] != file2_in_storage[i])

...

         s.add(storage_occupied[i]==
             Sum([If(file1_in_storage[f]==i, files[f], 0) for f in range(files_n)])+
             Sum([If(file2_in_storage[f]==i, files[f], 0) for f in range(files_n)]))

...

                 if m[file1_in_storage[f]].as_long()==i:
                     print "    file%d (1st copy) (%d)" % (f, files[f])
                 if m[file2_in_storage[f]].as_long()==i:
                     print "    file%d (2nd copy) (%d)" % (f, files[f])
```

```
...
```

(https://smt.st/current_tree/knapsack/backup/backup_twice.py)

The result:

```
storage total we need 3570
trying to fit all the files into storages: (0, 3, 5, 6, 10) sat
* solution (5 storages):
storage0 (total=700):
    file4 (1st copy) (167)
    file7 (1st copy) (368)
    file8 (1st copy) (144)
    file9 (2nd copy) (15)
allocated on storage=694 free on storage=6
storage3 (total=800):
    file0 (2nd copy) (18)
    file3 (1st copy) (184)
    file4 (2nd copy) (167)
    file6 (2nd copy) (45)
    file7 (2nd copy) (368)
    file9 (1st copy) (15)
allocated on storage=797 free on storage=3
storage5 (total=800):
    file0 (1st copy) (18)
    file1 (2nd copy) (57)
    file3 (2nd copy) (184)
    file5 (2nd copy) (496)
    file6 (1st copy) (45)
allocated on storage=800 free on storage=0
storage6 (total=300):
    file2 (1st copy) (291)
allocated on storage=291 free on storage=9
storage10 (total=1000):
    file1 (1st copy) (57)
    file2 (2nd copy) (291)
    file5 (1st copy) (496)
    file8 (2nd copy) (144)
allocated on storage=988 free on storage=12
total in all storages=3600
allocated on all storages=99%

trying to fit all the files into storages: (0, 3, 5, 9, 10) sat
* solution (5 storages):
storage0 (total=700):
    file0 (1st copy) (18)
    file3 (2nd copy) (184)
    file5 (2nd copy) (496)
allocated on storage=698 free on storage=2
storage3 (total=800):
    file1 (2nd copy) (57)
    file2 (2nd copy) (291)
    file4 (2nd copy) (167)
    file8 (1st copy) (144)
    file9 (2nd copy) (15)
allocated on storage=674 free on storage=126
storage5 (total=800):
    file4 (1st copy) (167)
    file6 (2nd copy) (45)
    file7 (1st copy) (368)
    file8 (2nd copy) (144)
allocated on storage=724 free on storage=76
storage9 (total=500):
    file2 (1st copy) (291)
    file3 (1st copy) (184)
allocated on storage=475 free on storage=25
storage10 (total=1000):
    file0 (2nd copy) (18)
    file1 (1st copy) (57)
    file5 (1st copy) (496)
    file6 (1st copy) (45)
    file7 (2nd copy) (368)
    file9 (1st copy) (15)
allocated on storage=999 free on storage=1
total in all storages=3800
allocated on all storages=93%
```

10.3 Packing virtual machines into servers

You've got these servers (all in GBs):

```
     RAM storage
 srv0  2     100
 srv1  4     800
 srv2  4    1000
 srv3 16    8000
 srv4  8    3000
 srv5 16    6000
 srv6 16    4000
 srv7 32    2000
 srv8  8    1000
 srv9 16   10000
srv10  8    1000
```

And you're going to put these virtual machines to servers:

```
    RAM storage
 VM0  1     100
 VM1 16     900
 VM2  4     710
 VM3  2     800
 VM4  4    7000
 VM5  8    4000
 VM6  2     800
 VM7  4    2500
 VM8 16     450
 VM9 16    3700
VM10 12    1300
```

The problem: use as small number of servers, as possible. Fit VMs into them in the most efficient way, so that the free RAM/storage would be minimal.

This is like knapsack problem. But the classic knapsack problem is about only one dimension (weight or size). We've two dimensions here: RAM and storage. This is called *multidimensional knapsack problem*.

Another problem we will solve here is a *bin packing problem*.

```
#!/usr/bin/env python3
from z3 import *
import itertools

# RAM, storage, both in GB
servers=[(2, 100),
(4, 800),
(4, 1000),
(16, 8000),
(8, 3000),
(16, 6000),
(16, 4000),
(32, 2000),
(8, 1000),
(16, 10000),
(8, 1000)]

# RAM, storage
vms=[(1, 100),
(16, 900),
(4, 710),
(2, 800),
(4, 7000),
(8, 4000),
(2, 800),
(4, 2500),
(16, 450),
(16, 3700),
(12, 1300)]

vms_total=len(vms)

VM_RAM_t=sum(map(lambda x: x[0], vms))
VM_storage_t=sum(map(lambda x: x[1], vms))

print ("total RAM we need", VM_RAM_t)
print ("total storage we need", VM_storage_t)

def try_to_fit_into_servers(servers_to_be_used):
    t=len(servers_to_be_used)
    # for each server:
    RAM_allocated=[Int('srv%d_RAM_allocated' % i) for i in range(t)]
    storage_allocated=[Int('srv%d_storage_allocated' % i) for i in range(t)]
    # which server this VM occupies?
    VM_in_srv=[Int('VM%d_in_srv' % i) for i in range(vms_total)]

    # how much RAM/storage we have in picked servers, total?
    RAM_t=0
    storage_t=0
    for i in range(t):
        RAM_t=RAM_t+servers[servers_to_be_used[i]][0]
        storage_t=storage_t+servers[servers_to_be_used[i]][1]
    # skip if the sum of RAM/storage in picked servers is too small:
    if VM_RAM_t>RAM_t or VM_storage_t>storage_t:
        return

    print ("trying to fit VMs into servers:", servers_to_be_used,end=' ')

    s=Solver()

    # all VMs must occupy some server:
    for i in range(vms_total):
        s.add(And(VM_in_srv[i]>=0, VM_in_srv[i]<t))

    for i in range(t):
        """
        here we generate expression like:

        If(VM0_in_srv == 3, 1, 0) +
        If(VM1_in_srv == 3, 16, 0) +
        If(VM2_in_srv == 3, 4, 0) +
        If(VM3_in_srv == 3, 2, 0) +
        If(VM4_in_srv == 3, 4, 0) +
        If(VM5_in_srv == 3, 8, 0) +
        If(VM6_in_srv == 3, 2, 0) +
        If(VM7_in_srv == 3, 4, 0) +
        If(VM8_in_srv == 3, 16, 0) +
        If(VM9_in_srv == 3, 16, 0) +
        If(VM10_in_srv == 3, 12, 0)

        ... in plain English - if a VM is in THIS server, add a number (RAM/storage
           required by this VM) to the final sum

        """

        # RAM
        s.add(RAM_allocated[i]==Sum([If(VM_in_srv[v]==i, vms[v][0], 0) for v in range
           (vms_total)]))
        # storage
        s.add(storage_allocated[i]==Sum([If(VM_in_srv[v]==i, vms[v][1], 0) for v in
           range(vms_total)]))

        # ... but sum of all RAM/storage occupied in each server must be lower than
           what we have in the server:
        s.add(And(RAM_allocated[i]>=0, RAM_allocated[i]<=servers[servers_to_be_used[i
           ]][0]))
        s.add(And(storage_allocated[i]>=0, storage_allocated[i]<=servers[
           servers_to_be_used[i]][1]))
    if s.check()==sat:
        print ("sat")
        print ("* solution (%d servers):" % t)
        m=s.model()

        for i in range(t):
            print ("srv%d (total=%d/%d):" % (servers_to_be_used[i], servers[
               servers_to_be_used[i]][0], servers[servers_to_be_used[i]][1]),end=' ')
            for v in range(vms_total):
                if m[VM_in_srv[v]].as_long()==i:
                    print ("VM%d (%d/%d)" % (v, vms[v][0], vms[v][1]),end=' ')
            print ("allocated on srv=%d/%d" % (m[RAM_allocated[i]].as_long(), m[
               storage_allocated[i]].as_long()),end=' ')
            print ("free on srv=%d/%d" % (servers[servers_to_be_used[i]][0] - m[
               RAM_allocated[i]].as_long(), servers[servers_to_be_used[i]][1] - m[
               storage_allocated[i]].as_long()),end=' ')
            print ("")
        print ("total in all servers=%d/%d" % (RAM_t, storage_t))
        print ("allocated on all servers=%d%%/%d%%" % ((float(VM_RAM_t)/float(RAM_t))
           *100, (float(VM_storage_t)/float(storage_t))*100))
        print ("")
        return True
    else:
        print ("unsat")
        return False

# how many servers we need? start with 2:
found_solution=False
for servers_to_pick in range(2, len(servers)+1):

    # we use Python itertools to find all combinations
    # in other words, pick $servers_to_pick$ servers from all servers, and enumerate
       all possible ways to choose from them.
    # see also: https://en.wikipedia.org/wiki/Combination
    for servers_to_be_used in itertools.combinations(range(len(servers)), r=servers_to_pick):
        if try_to_fit_into_servers(servers_to_be_used):
            found_solution=True
    # after we've got some solutions for $servers_to_pick$, stop:
    if found_solution:
        break
```

(https://smt.st/current_tree/knapsack/VM_packing/VM_pack.py)

The result:

```
RAM total we need 85
storage total we need 22260
trying to fit VMs into servers: (3, 4, 5, 6, 7) unsat
trying to fit VMs into servers: (3, 4, 5, 7, 9) sat
* solution (5 servers):
srv3 (total=16/8000): VM2 (4/710) VM3 (2/800) VM5 (8/4000) VM6 (2/800) allocated on srv=16/6310 free on srv=0/1690
srv4 (total=8/3000): VM0 (1/100) VM7 (4/2500) allocated on srv=5/2600 free on srv=3/400
srv5 (total=16/6000): VM9 (16/3700) allocated on srv=16/3700 free on srv=0/2300
srv7 (total=32/2000): VM1 (16/900) VM8 (16/450) allocated on srv=32/1350 free on srv=0/650
srv9 (total=16/10000): VM4 (4/7000) VM10 (12/1300) allocated on srv=16/8300 free on srv=0/1700
total in all servers=88/29000
allocated on all servers=96%/76%

trying to fit VMs into servers: (3, 4, 6, 7, 9) sat
* solution (5 servers):
srv3 (total=16/8000): VM2 (4/710) VM3 (2/800) VM5 (8/4000) VM6 (2/800) allocated on srv=16/6310 free on srv=0/1690
srv4 (total=8/3000): VM0 (1/100) VM7 (4/2500) allocated on srv=5/2600 free on srv=3/400
srv6 (total=16/4000): VM9 (16/3700) allocated on srv=16/3700 free on srv=0/300
srv7 (total=32/2000): VM1 (16/900) VM8 (16/450) allocated on srv=32/1350 free on srv=0/650
srv9 (total=16/10000): VM4 (4/7000) VM10 (12/1300) allocated on srv=16/8300 free on srv=0/1700
total in all servers=88/27000
allocated on all servers=96%/82%

trying to fit VMs into servers: (3, 5, 6, 7, 9) sat
* solution (5 servers):
srv3 (total=16/8000): VM0 (1/100) VM5 (8/4000) VM6 (2/800) VM7 (4/2500) allocated on srv=15/7400 free on srv=1/600
srv5 (total=16/6000): VM10 (12/1300) allocated on srv=12/1300 free on srv=4/4700
srv6 (total=16/4000): VM9 (16/3700) allocated on srv=16/3700 free on srv=0/300
srv7 (total=32/2000): VM1 (16/900) VM8 (16/450) allocated on srv=32/1350 free on srv=0/650
srv9 (total=16/10000): VM2 (4/710) VM3 (2/800) VM4 (4/7000) allocated on srv=10/8510 free on srv=6/1490
total in all servers=96/30000
allocated on all servers=88%/74%

trying to fit VMs into servers: (3, 5, 7, 8, 9) unsat
trying to fit VMs into servers: (3, 5, 7, 9, 10) unsat
trying to fit VMs into servers: (3, 6, 7, 8, 9) unsat
trying to fit VMs into servers: (3, 6, 7, 9, 10) unsat
trying to fit VMs into servers: (4, 5, 6, 7, 9) unsat
trying to fit VMs into servers: (5, 6, 7, 8, 9) unsat
trying to fit VMs into servers: (5, 6, 7, 9, 10) unsat
```

Choose any solution you like...

Further work: storage can be both HDD and/or SDD. That would add 3rd dimension. Or maybe number of CPU cores, network bandwidth, etc...

Chapter 11

Social Golfer Problem

> Twenty golfers wish to play in foursomes for 5 days. Is it possible for each golfer to play no more than once with any other golfer?
>
> ...
>
> Event organizers for bowling, golf, bridge, or tennis frequently tackle problems of this sort, unaware of the problem complexity. In general, it is an unsolved problem. A table of known results is maintained by Harvey.

(http://mathworld.wolfram.com/SocialGolferProblem.html)

It's also problem 010 in CSPLIB[1].

This is the solution that was ported from Markus Triska's solution written in Prolog[2].

This is an example solution for 7 golfers, 3 players in group, 9 weeks/days (7-3-9):

```
* week 1
[[1, 2, 4], [3, 5, 13], [6, 8, 19], [7, 16, 20], [9, 11, 14], [10, 15, 18], [12, 17,
    21]]
* week 2
[[1, 3, 10], [2, 11, 18], [4, 13, 16], [5, 14, 19], [6, 9, 12], [7, 15, 17], [8, 20,
    21]]
* week 3
[[1, 6, 14], [2, 7, 9], [3, 15, 21], [4, 10, 11], [5, 12, 20], [8, 13, 18], [16, 17,
    19]]
* week 4
[[1, 7, 11], [2, 5, 17], [3, 6, 16], [4, 9, 18], [8, 12, 15], [10, 19, 20], [13, 14,
    21]]
* week 5
[[1, 9, 20], [2, 12, 13], [3, 7, 8], [4, 6, 15], [5, 16, 18], [10, 14, 17], [11, 19,
    21]]
* week 6
[[1, 12, 19], [2, 16, 21], [3, 9, 17], [4, 7, 14], [5, 8, 10], [6, 18, 20], [11, 13,
    15]]
* week 7
[[1, 13, 17], [2, 6, 10], [3, 11, 12], [4, 5, 21], [7, 18, 19], [8, 9, 16], [14, 15,
    20]]
* week 8
[[1, 15, 16], [2, 3, 20], [4, 8, 17], [5, 6, 11], [7, 10, 21], [9, 13, 19], [12, 14,
    18]]
* week 9
[[1, 18, 21], [2, 8, 14], [3, 4, 19], [5, 9, 15], [6, 7, 13], [10, 12, 16], [11, 17,
    20]]
```

[1] https://csplib.github.io/csplib-builds/ghmeta/Problems/prob010/results/

[2] https://www.metalevel.at/sgpsat.pdf, https://www.metalevel.at/sgp/satgolf.pl

I also wanted to check timings when symmertry breaking constraints present and when not. These are my results. All checked on Intel(R) Xeon(R) CPU E3-1270 v3 @ 3.50GHz. Kissat 2.0.1 (2021) SAT solver was used. Time in seconds.

G-P-W	with symmetry breaking	without symmetry breaking
2-2-3	<1	<1
3-2-5	<1	<1
3-3-4	<1	<1
4-2-7	<1	<1
4-3-4	<1	<1
4-4-4	<1	<1
4-4-5	<1	-
5-2-9	<1	<1
6-2-11	<1	-
7-2-13	<1	1
5-3-7	1	29
5-4-5	1	<1
5-5-6	1	-
6-6-3	1	1
8-2-15	1	<1
9-2-17	1	1
6-4-5	2	<1
10-2-19	5	7
7-5-5	7	-
7-6-4	11	-
9-4-8	14	2649
9-5-6	14	75
9-8-3	15	4
8-7-4	16	1832
8-3-10	18	18089
9-9-3	19	4
9-7-4	26	23
9-3-11	33	5867
10-9-3	35	6
9-6-5	37	49
10-8-4	37	-
10-5-7	40	829
7-3-9	55	19147
10-4-9	81	21702
10-10-3	140	-
10-7-5	171	-
6-3-8	236	86172
10-6-6	393	4421
8-5-6	1423	881
10-3-13	2065	?
8-6-5	7105	689
7-4-7	11021	?
6-5-6	70026	66669
6-4-7	?	-
6-6-4	?	-
6-6-7	?	?
7-3-10	?	?
7-4-9	?	?
7-7-7	?	?
7-7-8	?	?
8-3-11	?	?
8-4-10	?	?
8-4-9	?	-
8-8-9	?	?
9-4-11	?	-

9-9-10	?	?

'?' — not solved withing 24 hours. '-' — not executed.

Clearly, almost always, symmetry breaking constraints are of help. But there are few cases, when they slows down everything.

Python source code: `https://smt.st/current_tree/SGP/gen.py`.

All resulting files: `https://smt.st/current_tree/SGP/results.tar`.

Further reading:

- Ian P. Gent and Inês Lynce – A SAT Encoding for the Social Golfer Problem.
- Markus Triska: Webpage, Solution Methods for the Social Golfer Problem.
- Warwick Harvey: Warwick's Results Page for the Social Golfer Problem, Solutions to various Social Golfer configurations.
- Ed Pegg Jr: Social Golfer Problem.
- `https://demonstrations.wolfram.com/SocialGolferProblem/`.
- Ke Liu, Sven Löffler and Petra Hofstedt: Solving the Social Golfers Problems by Constraint Programming in Sequential and Parallel.

Chapter 12

Latin squares

Magic/Latin square is a square filled with numbers/letters, which are all distinct in each row and column. Sudoku is 9*9 magic square with additional constraints (for each 3*3 subsquare).

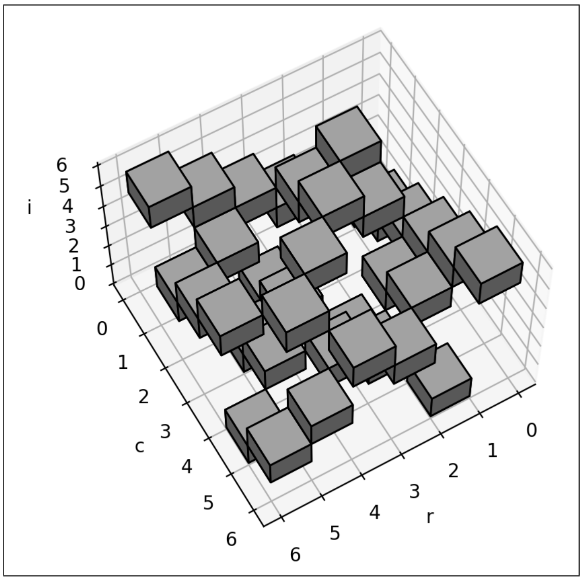
12.1 A simple Latin Square

A popular way of representing LS[1] as cube. Third dimension is added: symbol (number).

Sometimes also called "incidence cube" [see: Mark T. Jacobson, Peter Matthews – Generating Uniformly Distributed Random Latin Squares (1996)].

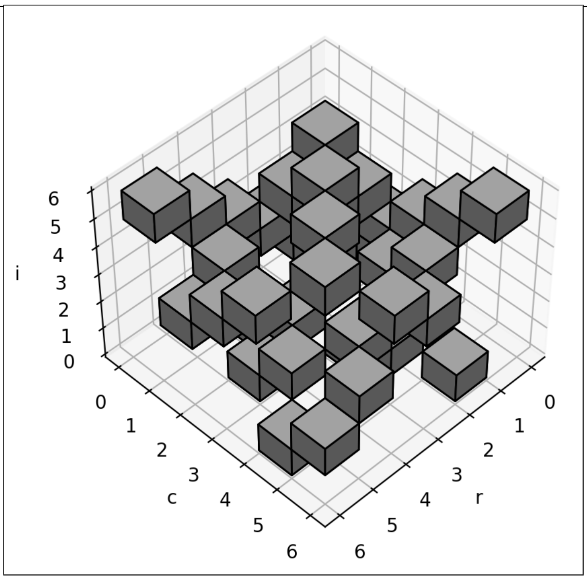
For example, this is the normalized LS [2] I visualized using matplotlib:

```
012345
154230
245013
301524
430152
523401
```

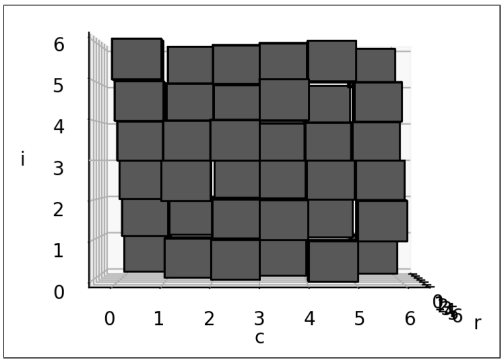


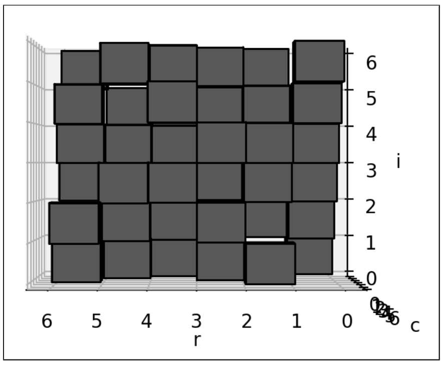
[1] Latin Square

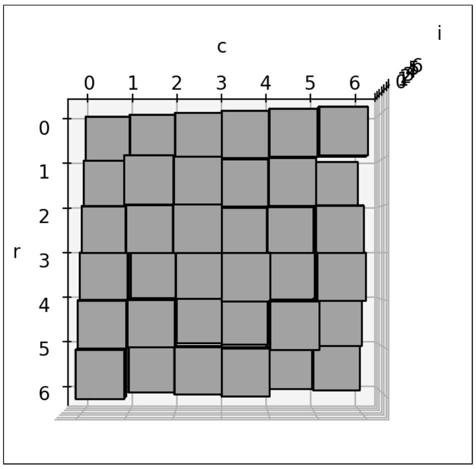
[2] A *Normalized/reduced* Latin square has first row and column consisting of ascending numbers/symbols (0,1,2,3...)



By looking from all three sides, you'll see all $6 \cdot 6 = 36$ cubelets, but only one in each *slot*:







Also, see the video for L(6) LS: https://www.youtube.com/watch?v=Fk-ysMqc098

See the video for L(10): https://www.youtube.com/watch?v=zT04G60VKD8. Notice the distinctive "V" shape, which is the result of normalization (0..9 in first row and column).

Also L(16): https://www.youtube.com/watch?v=Lovt41KVkwM, L(25): https://www.youtube.com/watch?v=1eVsFy_gLaY, L(36): https://www.youtube.com/watch?v=IgYuqRXry0A.

Adding constraints for such a *cube* is easy:

```
def latin_add_constraints(SIZE, s, ar):
    # 'cube'

    # one hot for each r, c.
    for r in range(SIZE):
        for c in range(SIZE):
            tmp=[ar[r][c][n] for n in range(SIZE)]
            s.make_one_hot(tmp)

    # one hot for each r, n
    for r in range(SIZE):
        for n in range(SIZE):
            tmp=[ar[r][c][n] for c in range(SIZE)]
            s.make_one_hot(tmp)

    # one hot for each c, n
    for c in range(SIZE):
        for n in range(SIZE):
            tmp=[ar[r][c][n] for r in range(SIZE)]
            s.make_one_hot(tmp)
```

(The full file: https://smt.st/current_tree/latin/latin_utils.py)

Adding ormalization constraints are also easy:

```
def fix_first_row_increasing(s, a, SIZE):
    # setting first row of $a$ to [0..SIZE-1]
    for _c in range(SIZE):
        _r=0
        tmp=SAT_lib.n_to_BV(1 << _c, SIZE)
        s.fix_BV(a[_r][_c], tmp)

def fix_first_col_increasing(s, a, SIZE):
    # setting all columns to [0..SIZE-1]
    for _r in range(SIZE):
        _c=0
        tmp=SAT_lib.n_to_BV(1 << _r, SIZE)
        s.fix_BV(a[_r][_c], tmp)
```

(The full file: https://smt.st/current_tree/latin/latin_utils.py)

Generating small LS is easy, see source code: https://smt.st/current_tree/latin/simple/LS.py.

The largest LS I could generate is of order 180 – 8 hours on AMD Ryzen 5 3600 6-Core Processor using the Gimsatul parallel SAT solver running on 6 threads: https://smt.st/current_tree/latin/simple/L180.txt.

An important note: forcing a Latin square to be *normalized* makes search space much smaller. Obviously, there are much less *normalized* squares than *usual* ones.

See a number of Latin squares in OEIS[3]:

```
A002860                 Number of Latin squares of order n; or labeled quasigroups.
1, 2, 12, 576, 161280, 812851200, 61479419904000, 108776032459082956800,
    5524751496156892842531225600, 9982437658213039871725064756920320000,
    776966836171770144107444346734230682311065600000
```

(https://oeis.org/A002860)

A number of *normalized/reduced* Latin squares:

[3] On-Line Encyclopedia of Integer Sequences

```
A000315                    Number of reduced Latin squares of order n; also number of
    labeled loops (quasigroups with an identity element) with a fixed identity element
    .
1, 1, 1, 4, 56, 9408, 16942080, 535281401856, 377597570964258816,
    7580721483160132811489280, 5363937773277371298119673540771840
```

(https://oeis.org/A000315)

And these is *symmetry breaking* constraints that makes Latin square to be *normalized/reduced*. Sometimes they make SAT solver work faster, but sometimes not. As they say, *your mileage may vary*, they can help for your specific problem, or may not.

12.2 Mutually orthogonal Latin squares: naive idea

AKA Graeco-Latin squares.

12.2.1 The principle

First I created a function that multiply two one-hot values. Which seemed to be very simple. To multiply x and y, create square where each variable would be True if both variables from x and y would be True at this intersection.

```
    def mult_one_hots(self, x:List[int], y:List[int]) -> List[int]:
        assert len(x)==len(y)
        in_size=len(x)
        z=[]
        for i in range(in_size):
            for j in range(in_size):
                z.append(self.AND(x[i], y[j]))
        assert len(z)==in_size**2
        return z
```

(The full file: https://smt.st/current_tree/libs/SAT_lib.py)

Using this function, I can join two LS into MOLS:

```
def make_mutually_orthogonal(SIZE, s, a, b):
    t=[]
    for r in range(SIZE):
        for c in range(SIZE):
            t.append(s.mult_one_hots(a[r][c], b[r][c]))

    for x in my_utils.transpose_matrix(t):
        s.OR_always(x)
```

(The full file: https://smt.st/current_tree/latin/latin_utils.py)

Each pair must be present in the final solution, hence you see OR_always() at the end.

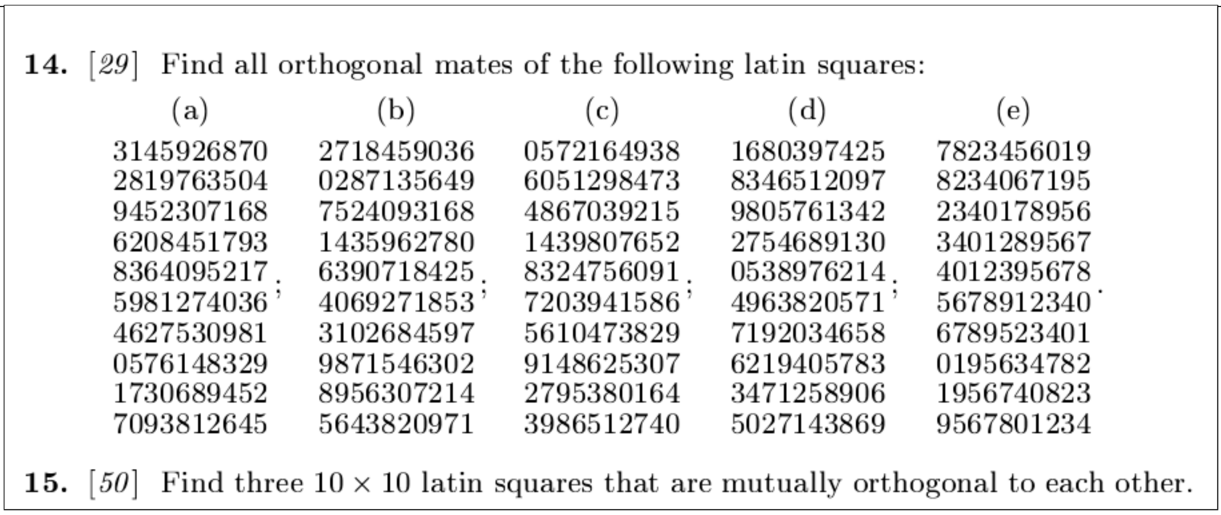
The full source code: https://smt.st/current_tree/latin/MOLS_naive/MOLS2_par.py.

In reasonable time, it can find: 2-MOLS(7): https://smt.st/current_tree/latin/MOLS_naive/2-MOLS7.txt; 2-MOLS(8): https://smt.st/current_tree/latin/MOLS_naive/2-MOLS8.txt; 2-MOLS(9): https://smt.st/current_tree/latin/MOLS_naive/2-MOLS9.txt; 2-MOLS(10): https://smt.st/current_tree/latin/MOLS_naive/2-MOLS10.txt. Using Gimsatul (10 threads) on AMD Ryzen 5 3600 6-Core Processor.

12.2.2 Donald Knuth's exercise

We will try to solve this problem from Donald Knuth – TAOCP – Introduction to combinatorial searching [4]:

[4] https://yurichev.com/mirrors/Donald%20Knuth/TAOCP%200a%207/fasc0a.pdf

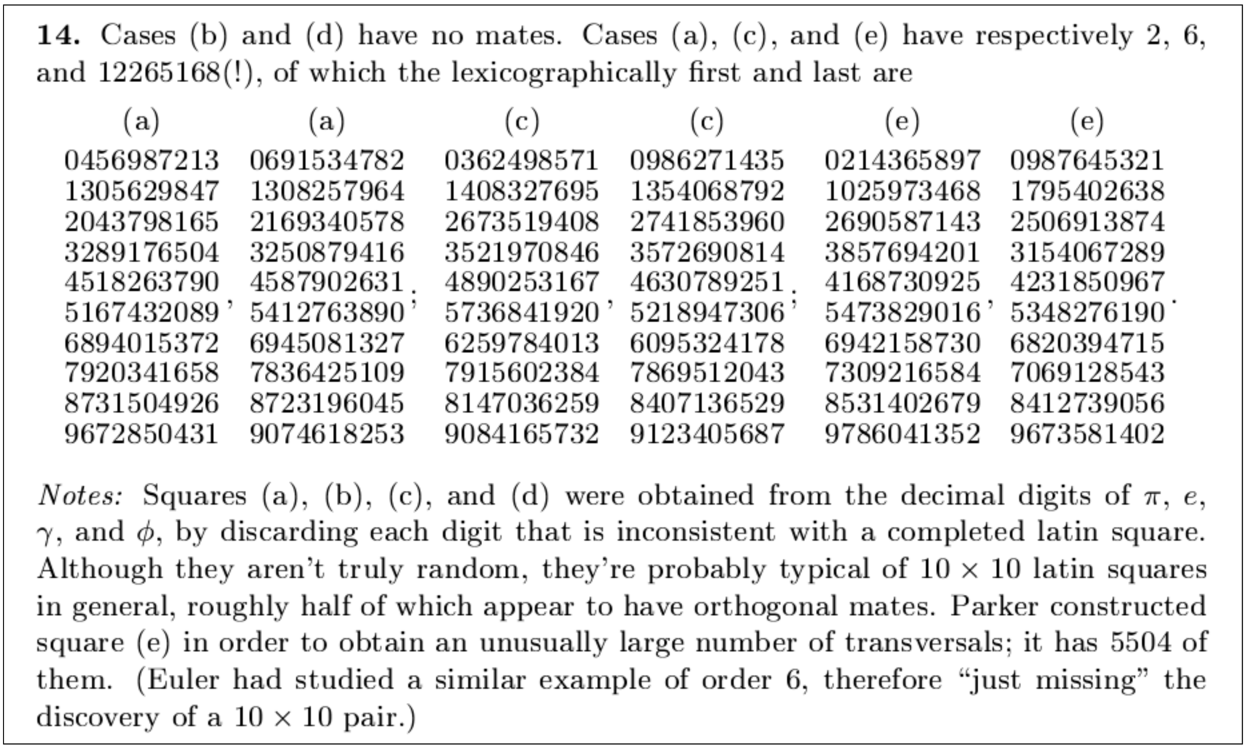


14. [*29*] Find all orthogonal mates of the following latin squares:

(a)	(b)	(c)	(d)	(e)
3145926870	2718459036	0572164938	1680397425	7823456019
2819763504	0287135649	6051298473	8346512097	8234067195
9452307168	7524093168	4867039215	9805761342	2340178956
6208451793	1435962780	1439807652	2754689130	3401289567
8364095217 ;	6390718425 ;	8324756091 ;	0538976214 ;	4012395678 .
5981274036	4069271853	7203941586	4963820571	5678912340
4627530981	3102684597	5610473829	7192034658	6789523401
0576148329	9871546302	9148625307	6219405783	0195634782
1730689452	8956307214	2795380164	3471258906	1956740823
7093812645	5643820971	3986512740	5027143869	9567801234

15. [*50*] Find three 10 × 10 latin squares that are mutually orthogonal to each other.

Figure 12.1: The problem



14. Cases (b) and (d) have no mates. Cases (a), (c), and (e) have respectively 2, 6, and 12265168(!), of which the lexicographically first and last are

(a)	(a)	(c)	(c)	(e)	(e)
0456987213	0691534782	0362498571	0986271435	0214365897	0987645321
1305629847	1308257964	1408327695	1354068792	1025973468	1795402638
2043798165	2169340578	2673519408	2741853960	2690587143	2506913874
3289176504	3250879416	3521970846	3572690814	3857694201	3154067289
4518263790 ,	4587902631 ;	4890253167 ,	4630789251 ;	4168730925 ,	4231850967 .
5167432089	5412763890	5736841920	5218947306	5473829016	5348276190
6894015372	6945081327	6259784013	6095324178	6942158730	6820394715
7920341658	7836425109	7915602384	7869512043	7309216584	7069128543
8731504926	8723196045	8147036259	8407136529	8531402679	8412739056
9672850431	9074618253	9084165732	9123405687	9786041352	9673581402

Notes: Squares (a), (b), (c), and (d) were obtained from the decimal digits of π, e, γ, and ϕ, by discarding each digit that is inconsistent with a completed latin square. Although they aren't truly random, they're probably typical of 10 × 10 latin squares in general, roughly half of which appear to have orthogonal mates. Parker constructed square (e) in order to obtain an unusually large number of transversals; it has 5504 of them. (Euler had studied a similar example of order 6, therefore "just missing" the discovery of a 10 × 10 pair.)

Figure 12.2: The solution

I'm fixing solution's column to be in ascending order (0,1,2,3...) like in D.Knuth's solutions: solutions count would be multiple of 10! otherwise (number of isomorphic squares).

Listing 12.1: Commands

```
# 2 solutions
# Knuth_a
    ="3145926870281976350494523071686208451793836409521759812740364627530981057614832917306894527093812645"

my_time.py ./MOLS2_par.py --solver gimsatul --all-solutions --normalize --first
3145926870281976350494523071686208451793836409521759812740364627530981057614832917306894527093812645

# 6 solutions
# Knuth_c
    ="0572164938605129847348670392151439807652832475609172039415865610473829914862530727953801643986512740"

my_time.py ./MOLS2_par.py --solver gimsatul --all-solutions --normalize --first
0572164938605129847348670392151439807652832475609172039415865610473829914862530727953801643986512740
```

```
# 12,265,168 solutions
# Knuth_e
    ="782345601982340671952340178956340128956740123956785678912340678952340101956347821956740823956780123 4"

my_time.py ./MOLS2_par.py --solver gimsatul --all-solutions --normalize --first
7823456019823406719523401789563401289567401239567856789123406789523401019563478219567408239567801234
```

2 solutions for (a): https://smt.st/current_tree/latin/MOLS_naive/TAOCP_a.txt

6 solutions for (c): https://smt.st/current_tree/latin/MOLS_naive/TAOCP_c.txt

Some solutions for (e) (not all): https://smt.st/current_tree/latin/MOLS_naive/TAOCP_e.txt

All timings: Gimsatul using 10 threads running on AMD Ryzen 5 3600 6-Core Processor.

12.2.3 Proofs

Thirty-six officers problem is a well-known classic problem. Citing Wikipedia:

> A problem similar to the card problem above was circulating in St. Petersburg in the late 1700s and, according to folklore, Catherine the Great asked Euler to solve it, since he was residing at her court at the time.[7] This problem is known as the thirty-six officers problem,[8] and Euler introduced it as follows:[9][10]
>
> A very curious question, which has exercised for some time the ingenuity of many people, has involved me in the following studies, which seem to open a new field of analysis, in particular the study of combinations. The question revolves around arranging 36 officers to be drawn from 6 different regiments so that they are ranged in a square so that in each line (both horizontal and vertical) there are 6 officers of different ranks and different regiments. — Leonhard Euler

(https://en.wikipedia.org/wiki/Mutually_orthogonal_Latin_squares)

It was proved by Gaston Tarry in 1901. But we can prove it as well.

In other words, we will try to find a 2-MOLS(6), but this is impossible, my script would report UNSAT. But can we create a proof?

```
my_time.py ./MOLS2_par.py --solver gimsatul --normalize --proof --size 6
```

The proof file size: ≈ 31 MiB.

However, when I tried to find a proof for non-normalized LS, things are bad. Gimsatul was running for more than 14 hours and it produced already ≈ 200 GiB proof, so I had to interrupt it.

Presumably, the problem in much larger search space. There are just 9,408 reduced L(6) (OEIS: A000315), but 812,851,200 non-reduced L(6) (OEIS: A002860).

It's also possible to find proofs for two L(10) from TAOCP that has no MOLS:

```
# Knuth_b
my_time.py ./MOLS2_par.py --solver gimsatul --all-solutions --normalize --proof --first
2718459036028713564975240931681435962780639071842540692718533102684597987154630289563072145643820971
```

≈ 5 minutes using Gimsatul on 10 cores. The proof size: ≈ 600 MiB.

```
# Knuth_d
my_time.py ./MOLS2_par.py --solver gimsatul --all-solutions --normalize --proof --first
1680397425834651209798057613422754689130053897621449638205717192034658621940578334712589065027143869
```

≈ 5 minutes using Gimsatul on 10 cores. The proof size: ≈ 600 MiB.

12.2.4 Finding 3-MOLS

Almost the same, but 3 LS, all 3 pairs are orthogonal to each other. https://smt.st/current_tree/latin/MOLS_naive/MOLS3_par.py.

In reasonable time, it can find: 3-MOLS(7): https://smt.st/current_tree/latin/MOLS_naive/3-MOLS7.txt; 3-MOLS(8): https://smt.st/current_tree/latin/MOLS_naive/3-MOLS8.txt. Using Gimsatul (10 threads) on AMD Ryzen 5 3600 6-Core Processor.

Of course, finding 3-MOLS(10) is not (yet?) possible.

Finding 3 Latin squares of order 10 is a problem marked as having maximal (50) difficulty in TAOCP. [See the problem 15: 12.1.] A discussion on the Stack Exchange:
https://math.stackexchange.com/questions/649548/find-three-10-times10-orthogonal-latin-squares.

3-MOLS(9) is also out of reach.

However, we can prove that there is no 3-MOLS(10) for specific LS. Let's take L(10) (a) from TAOCP, for which 2 MOLS exists.

```
my_time.py ./MOLS3_par.py --solver gimsatul --all-solutions --normalize --proof --first
3145926870281976350494523071686208451793836409521759812740364627530981057614832917306894527093812645
```

The proof file is ≈ 1 GiB.

I tried to do the same for L(10) (e) LS, but that is out of reach, because (e) LS has 12,265,168 MOLS and search space is much larger.

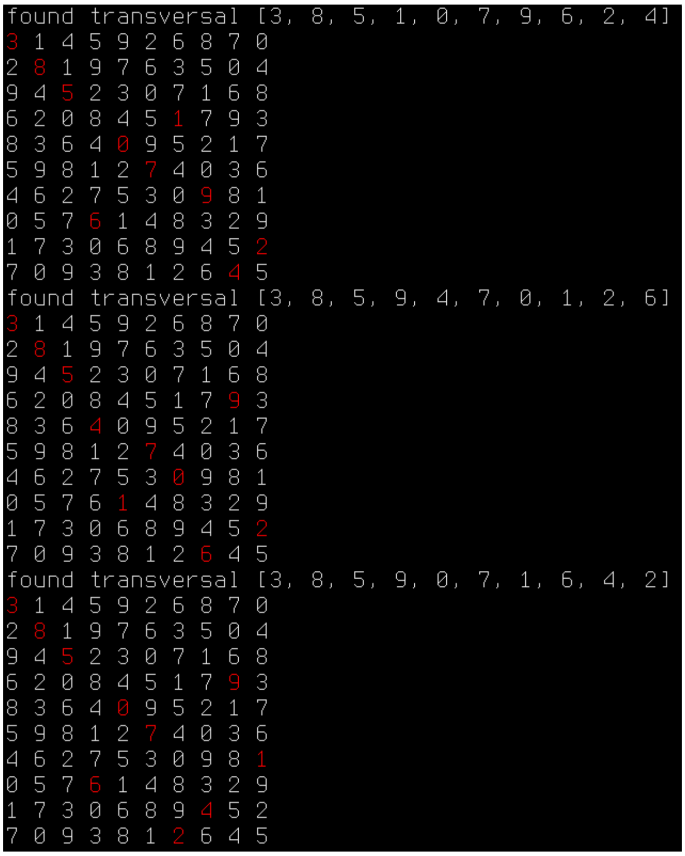
Of course, checking all L(10) (even narrowed to reduced/normalized, isotopy/main classes) is not possible [5]. There are 34,817,397,894,749,939 main classes for L(10).

12.3 Mutually orthogonal Latin squares: finding via transversals

This is the method mentioned in Donald Knuth – TAOCP – Introduction to combinatorial searching [6]. Named by D.Knuth as Euler-Parker method.

First, find all transversals in LS. Transversal is a set of squares/symbols. All symbols must be differernt in transversal. All rows and columns must be also different.

These are first 3 transversals of L(10) (a) from TAOCP:



```
found transversal [3, 8, 5, 1, 0, 7, 9, 6, 2, 4]
3 1 4 5 9 2 6 8 7 0
2 8 1 9 7 6 3 5 0 4
9 4 5 2 3 0 7 1 6 8
6 2 0 8 4 5 1 7 9 3
8 3 6 4 0 9 5 2 1 7
5 9 8 1 2 7 4 0 3 6
4 6 2 7 5 3 0 9 8 1
0 5 7 6 1 4 8 3 2 9
1 7 3 0 6 8 9 4 5 2
7 0 9 3 8 1 2 6 4 5
found transversal [3, 8, 5, 9, 4, 7, 0, 1, 2, 6]
3 1 4 5 9 2 6 8 7 0
2 8 1 9 7 6 3 5 0 4
9 4 5 2 3 0 7 1 6 8
6 2 0 8 4 5 1 7 9 3
8 3 6 4 0 9 5 2 1 7
5 9 8 1 2 7 4 0 3 6
4 6 2 7 5 3 0 9 8 1
0 5 7 6 1 4 8 3 2 9
1 7 3 0 6 8 9 4 5 2
7 0 9 3 8 1 2 6 4 5
found transversal [3, 8, 5, 9, 0, 7, 1, 6, 4, 2]
3 1 4 5 9 2 6 8 7 0
2 8 1 9 7 6 3 5 0 4
9 4 5 2 3 0 7 1 6 8
6 2 0 8 4 5 1 7 9 3
8 3 6 4 0 9 5 2 1 7
5 9 8 1 2 7 4 0 3 6
4 6 2 7 5 3 0 9 8 1
0 5 7 6 1 4 8 3 2 9
1 7 3 0 6 8 9 4 5 2
7 0 9 3 8 1 2 6 4 5
```

More transversals for that L(10): https://www.youtube.com/watch?v=8pDCTNvgOmE.

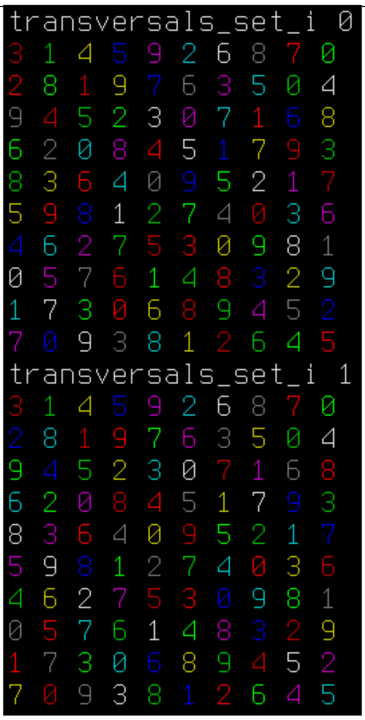
On average, L(10) has ≈ 800 transversals.

Then find such a set of 10 transversals, so that L(10) will be covered and each square will be covered by one transversal. This is no different from pentomino tiling problem 22.7 and solved using exact cover problem.

For L(10) (a) from TAOCP there are two ways of covering (each transversal colored by each color):

[5] At OEIS, see: A002860, A000315, A040082, A003090.

[6] https://yurichev.com/mirrors/Donald%20Knuth/TAOCP%200a%207/fasc0a.pdf



```
transversals_set_i 0
3 1 4 5 9 2 6 8 7 0
2 8 1 9 7 6 3 5 0 4
9 4 5 2 3 0 7 1 6 8
6 2 0 8 4 5 1 7 9 3
8 3 6 4 0 9 5 2 1 7
5 9 8 1 2 7 4 0 3 6
4 6 2 7 5 3 0 9 8 1
0 5 7 6 1 4 8 3 2 9
1 7 3 0 6 8 9 4 5 2
7 0 9 3 8 1 2 6 4 5
transversals_set_i 1
3 1 4 5 9 2 6 8 7 0
2 8 1 9 7 6 3 5 0 4
9 4 5 2 3 0 7 1 6 8
6 2 0 8 4 5 1 7 9 3
8 3 6 4 0 9 5 2 1 7
5 9 8 1 2 7 4 0 3 6
4 6 2 7 5 3 0 9 8 1
0 5 7 6 1 4 8 3 2 9
1 7 3 0 6 8 9 4 5 2
7 0 9 3 8 1 2 6 4 5
```

(L(10) (e) from TAOCP has 5504 transversals [7]. Here is also a video where these transversals are combined: https://www.youtube.com/watch?v=ifvKt_ts8DE.)

Then add the following constraint: second square (or mate): each transversal in mate must have same number/symbol.

This divides the task to 3 subtasks: 1) find all transversals in LS (fast); 2) combine *order* transversals using exact cover problem (toughest step); 3) find mate for transversals set (*order*! permutations).

That method was used in 1959 by E.T.Parker. Citing Wikipedia:

> Then E. T. Parker found a counterexample of order 10 using a one-hour computer search on a UNIVAC 1206 Military Computer while working at the UNIVAC division of Remington Rand (this was one of the earliest combinatorics problems solved on a digital computer).

(https://en.wikipedia.org/wiki/Mutually_orthogonal_Latin_squares)

The famous magazine cover:

[7] By the way, that square visualized as a cube: https://www.youtube.com/watch?v=gnNq-J5kcl4. Clearly, you can see here some symmetries/regularities.

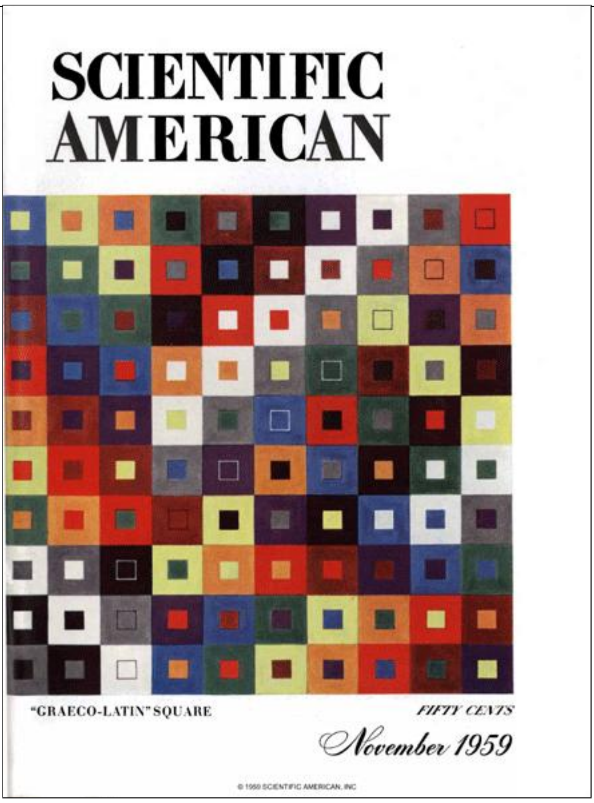


Figure 12.3: Scientific American, November 1959

(See also: http://www.martin-gardner.org/SciAm12.html.)

Of course, this method is way more faster than my first *naive* idea. But more sophisticated.

Here I found mates for TAOCP squares. (a): https://smt.st/current_tree/latin/MOLS_transv/TAOCP_a.txt; (c): https://smt.st/current_tree/latin/MOLS_transv/TAOCP_c.txt; some mates for (e): https://smt.st/current_tree/latin/MOLS_transv/TAOCP_e.txt.

The source code: https://smt.st/current_tree/latin/MOLS_transv/MOLS.py. Some transversals code is in: https://smt.st/current_tree/latin/latin_utils.py.

Using this method I can find 2-MOLS(11): https://smt.st/current_tree/latin/MOLS_transv/2-MOLS11.txt, 2-MOLS(12): https://smt.st/current_tree/latin/MOLS_transv/2-MOLS12.txt, 2-MOLS(13): https://smt.st/current_tree/latin/MOLS_transv/2-MOLS13.txt. (All timings are for AMD Ryzen 5 3600.)

I had no success in finding 2-MOLS(14), for days.

The proofs of MOLS absence are possible as well:

```
# no solutions
# Knuth_b
    ="2718459036028713564975240931681435962780639071842540692718533102684597987154630289563072145643820971"

my_time.py ./MOLS.py --proof --first
2718459036028713564975240931681435962780639071842540692718533102684597987154630289563072145643820971
```

The proof file filesize: ≈ 47MiB.

```
# no solutions
# Knuth_d
    ="1680397425834651209798057613422754689130053897621449638205717192034658621940578334712589065027143869"

my_time.py ./MOLS.py --proof --first
1680397425834651209798057613422754689130053897621449638205717192034658621940578334712589065027143869
```

The proof file filesize: ≈ 49MiB.

But this proof is different from proofs generated by my *naive* version. These are proofs of inexistance of solutions for exact cover problem. Which are, of course, the same as proofs of inexistance of mates for these LS.

However, finding 3-MOLS(n) using this method is not as good as my *naive* method, described previously. The *naive* method can find 3-MOLS(8) easily, but this method is not competitive.

Probably because the *naive* method find triple *in parallel*.

While this method do this sequentially: it finds a first LS, then tries to find a LS that is MOLS for the first LS, then tries to find a third LS which is MOLS for first and second LS. That is 3 stages.

12.4 Practical use of Latin squares

> For example, it turns out that orthogonal latin squares are enormously useful, particularly in the design of experiments. Already in 1788, Francois Crette de Palluel used a 4x4 latin square to study what happens when sixteen sheep - four each from four different breeds - were fed four different diets and harvested at four different times. [Memoires d'Agriculture (Paris: Societe Royale d'Agriculture, trimestre d'ete, 1788), 17-23.] The latin square allowed him to do this with 16 sheep instead of 64; with a Greco-Latin square he could also have varied another parameter by trying, say, four different quantities of food or four different grazing paradigms.
> But if we had focused our discussion on his approach to animal husbandry, we might well have gotten bogged down in details about breeding, about root vegetables versus grains and the costs of growing them, etc. Readers who aren't farmers might therefore have decided to skip the whole topic, even though latin square designs apply to a wide range of studies. (Think about testing five kinds of pills, on patients in five stages of some disease, five age brackets, and five weight groups.) Moreover, a concentration on experimental design could lead readers to miss the fact that latin squares also have important applications to coding and cryptography (see exercises 18-24).

(Donald Knuth – TAOCP – Introduction to combinatorial searching [8].)

Maybe you got a program you want to test against different compilers, different optimization options, etc. Maybe this program is CPU intensive app, like Bitcoin miner, SETI@home client or whatever. And you're unsure, which compiler would produce the best code.

Say, you have 4 compilers, 4 optimization options and 4 computers to test everything. Full test will take one day. You can arrange all this like:

Day	Computer 1 Compiler 1	Computer 2 Compiler 2	Computer 3 Compiler 3	Computer 4 Compiler 4
Day 1	Opt.options 1	Opt.options 2	Opt.options 3	Opt.options 4
Day 2	Opt.options 2	Opt.options 3	Opt.options 4	Opt.options 1
Day 3	Opt.options 3	Opt.options 4	Opt.options 1	Opt.options 2
Day 4	Opt.options 4	Opt.options 1	Opt.options 2	Opt.options 3

After testing, you'll find a best pair of compiler and optimization options.

This is a Latin square. You saw it often in Sudoku form, which is also Latin square, but with additional constraints (9 3*3 boxes).

Thanks to Latin square, you could test all pairs in 4 days using only 4 computers. Even if you have no idea about Latin squares and will try to find this arrangement manually, you'll come with a solution similar to that, because there is no smaller solution.

Now the problem is harder: you have 4 frameworks/libraries/APIs to test. Finally, you want to find a best triple: compiler + optimization options + framework.

For the problem like that it would be hard to find a good arrangement manually. I'm using my small utility to find two Latin squares, that are mutually orthogonal to each other:

```
first:    | mate:
0 1 2 3   | 0 3 1 2
1 0 3 2   | 1 2 0 3
2 3 0 1   | 2 1 3 0
3 2 1 0   | 3 0 2 1

concatenated:
00 13 21 32
11 02 30 23
22 31 03 10
33 20 12 01
```

[8] https://yurichev.com/mirrors/Donald%20Knuth/TAOCP%200a%207/fasc0a.pdf

Each pair in the final 'concatenated' (or superimposed) table is unique. Now add this 'concatenated' table to ours:

Day	Computer 1 Compiler 1	Computer 2 Compiler 2	Computer 3 Compiler 3	Computer 4 Compiler 4
Day 1	0+0	1+3	2+1	3+2
Day 2	1+1	0+2	3+0	2+3
Day 3	2+2	3+1	0+3	1+0
Day 4	3+3	2+0	1+2	0+1

Each pair of digits would reflect optimization options + framework. Thanks to MOLS (Mutually Orthogonal Latin Squares) you can test all triplets (compiler + optimization options + framework) (again) in 4 days using 4 computers.

Now the next problem: you also have 4 different operating systems. macOS, Windows, and maybe two flavours of Unix. 3 MOLS would help.

```
first:    | mate 1:   | mate 2:
0 1 2 3   | 0 2 3 1   | 0 3 1 2
1 0 3 2   | 1 3 2 0   | 1 2 0 3
2 3 0 1   | 2 0 1 3   | 2 1 3 0
3 2 1 0   | 3 1 0 2   | 3 0 2 1

first+mate 1 | first+mate 2 | mate 1+mate 2
00 12 23 31  | 00 13 21 32  | 00 23 31 12
11 03 32 20  | 11 02 30 23  | 11 32 20 03
22 30 01 13  | 22 31 03 10  | 22 01 13 30
33 21 10 02  | 33 20 12 01  | 33 10 02 21

first+mate 1+mate2:
000 123 231 312
111 032 320 203
222 301 013 130
333 210 102 021
```

These are 3 Latin squares where any pair of these 3 squares are orthogonal to each other. Hence, in the final table you see only unique triplets. Let's add this to our table:

Day	Computer 1 Compiler 1	Computer 2 Compiler 2	Computer 3 Compiler 3	Computer 4 Compiler 4
Day 1	0+0+0	1+2+3	2+3+1	3+1+2
Day 2	1+1+1	0+3+2	3+2+0	2+0+3
Day 3	2+2+2	3+0+1	0+1+3	1+3+0
Day 4	3+3+3	2+1+0	1+0+2	0+2+1

Each unique triplet is: optimization options + framework + OS. Again you need only 4 days and 4 computers to test each pair of these parameters and pick the best set of compiler + optimization options + framework + OS.

Unfortunately, MOLS generation has its limits too. Only small squares of order less than 10 are easy to generate. Generation of larger is a very hard problem. Despite of that, small MOLS are very useful.

12.5 Further reading

- Donald Knuth – TAOCP – Introduction to combinatorial searching [9].
- Martin Gardner – New Mathematical Diversions, 1995, pp 162-172.
- Combinatorics: Ancient & Modern (2013), Lars Døvling Andersen – Latin Squares, pp 251-283.
- Hantao Zhang – Combinatorial Designs by SAT Solvers[10].

[9] https://yurichev.com/mirrors/Donald%20Knuth/TAOCP%200a%207/fasc0a.pdf

[10] https://www.researchgate.net/publication/242029072_Combinatorial_Designs_by_SAT_Solvers, https://ebooks.iospress.nl/volumearticle/5002, https://ebooks.iospress.nl/volumearticle/56529.

Chapter 13

Cyclic redundancy check

My Untitled collection of math notes[1]has a yet another explanation of CRC.

13.1 Hacking CRC64 with KLEE

All timings are when KLEE 3.0 running on AMD Ryzen 5 3600.

13.1.1 Buffer alteration case #1

Sometimes, you need to alter a piece of data which is *protected* by some kind of checksum or CRC[2], and you can't change checksum or CRC value, but can alter piece of data so that checksum will remain the same.

Let's pretend, we've got a piece of data with "Hello, world!" string at the beginning and "and goodbye" string at the end. We can alter 14 characters at the middle, but for some reason, they must be in *a..z* limits, but we can put any characters there. CRC64 of the whole block must be `0x12345678abcdef12`.

Let's see[3]:

```
#include <string.h>
#include <stdint.h>

#include "klee.h"

// CRC-64-ECMA in https://en.wikipedia.org/wiki/Cyclic_redundancy_check
uint64_t crc64(uint64_t crc, unsigned char *buf, int len)
{
        int k;

        crc = ~crc;
        while (len--)
        {
                crc ^= *buf++;
                for (k = 0; k < 8; k++)
                        crc = crc & 1 ? (crc >> 1) ^ 0xC96C5795D7870F42 : crc >> 1;
        }
        return ~crc;
}

int main()
{
#define HEAD_STR "Hello, world!.. "
#define HEAD_SIZE strlen(HEAD_STR)
#define TAIL_STR " ... and goodbye"
#define TAIL_SIZE strlen(TAIL_STR)
#define MID_SIZE 14 // stuff
//#define MID_SIZE 13 // (failed) stuff
```

[1] https://xn--mxacd.xn--qxam/math-notes.pdf

[2] Cyclic redundancy check

[3] There are several slightly different CRC64 implementations, the one I use here can also be different from popular ones.

```
#define BUF_SIZE HEAD_SIZE+TAIL_SIZE+MID_SIZE

        char buf[BUF_SIZE];

        klee_make_symbolic(buf, sizeof buf, "buf");

        klee_assume (memcmp (buf, HEAD_STR, HEAD_SIZE)==0);

        for (int i=0; i<MID_SIZE; i++)
                klee_assume (buf[HEAD_SIZE+i]>='a' && buf[HEAD_SIZE+i]<='z');

        klee_assume (memcmp (buf+HEAD_SIZE+MID_SIZE, TAIL_STR, TAIL_SIZE)==0);

        klee_assume (crc64 (0, buf, BUF_SIZE)==0x12345678abcdef12);
}
```

We got 3 unit tests, the last one is correct:

```
% clang -emit-llvm -c -g -O0 -Xclang -disable-O0-optnone -I /tmp/klee_src/include/
   klee/ klee_CRC64.c

% time klee --libc=uclibc --optimize klee_CRC64.bc

...

KLEE: done: total instructions = 17036
KLEE: done: completed paths = 1
KLEE: done: partially completed paths = 32
KLEE: done: generated tests = 3

real    2m47.538s
user    2m46.299s
sys     0m0.619s

% ls klee-last/*
klee-last/assembly.ll   klee-last/solver-queries.smt2  klee-last/test000002.ktest
klee-last/info          klee-last/test000001.kquery    klee-last/test000002.user.err
klee-last/messages.txt  klee-last/test000001.ktest     klee-last/test000003.ktest
klee-last/run.istats    klee-last/test000001.user.err  klee-last/warnings.txt
klee-last/run.stats     klee-last/test000002.kquery

% ktest-tool klee-last/test000001.ktest | grep data
object 0: data: b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
   x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
   x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'

% ktest-tool klee-last/test000002.ktest | grep data
object 0: data: b'Hello, world!.. aaaaaaaaaaaaaa\x00\xff\xff\xff\xff\xff\xff\xff\xff\
   xff\xff\xff\xff\xff\xff\xff'

% ktest-tool klee-last/test000003.ktest | grep data
object 0: data: b'Hello, world!.. xnzhtcktymchxy ... and goodbye'
```

Since our code uses memcmp() standard C/C++ function, we need to add `--libc=uclibc` switch, so KLEE will use its own uClibc implementation.

Maybe it's slow, but definitely faster than bruteforce. Indeed, $log_2 26^{14} \approx 65.8$ which is close to 64 bits. In other words, one need ≈ 14 latin characters to encode 64 bits. And KLEE + SMT solver needs 64 bits at some place it can alter to make final CRC64 value equal to what we defined.

I tried to reduce length of the *middle block* to 13 characters: no luck for KLEE then, it has no space enough.

13.1.2 Buffer alteration case #2

I went sadistic: what if the buffer must contain the CRC64 value which, after calculation of CRC64, will result in the same value? Fascinatingly, KLEE can solve this. The buffer will have the following format:

```
Hello, world!.. <8 bytes (64-bit value)> <6 more bytes> ... and goodbye
```

Listing 13.1: Diff

```
--- klee_CRC64.c        2023-12-03 01:50:06.613088412 +0200
+++ klee_CRC64_v2.c     2023-12-03 01:36:56.673299339 +0200
@@ -24,8 +24,8 @@
 #define HEAD_SIZE strlen(HEAD_STR)
 #define TAIL_STR " ... and goodbye"
 #define TAIL_SIZE strlen(TAIL_STR)
-#define MID_SIZE 14 // stuff
-//#define MID_SIZE 13 // (failed) stuff
+// first 8 bytes is a 64-bit value:
+#define MID_SIZE 14
 #define BUF_SIZE HEAD_SIZE+TAIL_SIZE+MID_SIZE

        char buf[BUF_SIZE];
@@ -34,11 +34,10 @@

         klee_assume (memcmp (buf, HEAD_STR, HEAD_SIZE)==0);

-        for (int i=0; i<MID_SIZE; i++)
-                klee_assume (buf[HEAD_SIZE+i]>='a' && buf[HEAD_SIZE+i]<='z');
-
         klee_assume (memcmp (buf+HEAD_SIZE+MID_SIZE, TAIL_STR, TAIL_SIZE)==0);

-        klee_assume (crc64 (0, buf, BUF_SIZE)==0x12345678abcdef12);
+        uint64_t mid_value=*(uint64_t*)(buf+HEAD_SIZE);
+
+        klee_assume (crc64 (0, buf, BUF_SIZE)==mid_value);
 }
```

It works:

```
% time klee --libc=uclibc --optimize klee_CRC64_v2.bc
...
KLEE: done: total instructions = 16901
KLEE: done: completed paths = 1
KLEE: done: partially completed paths = 32
KLEE: done: generated tests = 3

real    2m4.957s
user    2m4.803s

% ls klee-last/*
klee-last/assembly.ll   klee-last/solver-queries.smt2  klee-last/test000002.ktest
klee-last/info          klee-last/test000001.kquery    klee-last/test000002.user.err
klee-last/messages.txt  klee-last/test000001.ktest     klee-last/test000003.ktest
klee-last/run.istats    klee-last/test000001.user.err  klee-last/warnings.txt
klee-last/run.stats     klee-last/test000002.kquery

% ktest-tool klee-last/test000003.ktest | grep data
object 0: data: b'Hello, world!.. \r\xc6\x98\x14\x05\xd2X\xef\xc2^\r\x01\x1b\xc9 ...
   and goodbye'
```

8 bytes between two strings is 64-bit value which equals to CRC64 of this whole block itself: 0xef58d2051498c60d. Also, there are 14-8=6 bytes after that 64-bit value which seems random, but it's OK. Again, it's faster than brute-force way to find it. If to decrease last spare 6-byte buffer to 4 bytes or less, KLEE works so long so I've stopped it.

13.1.3 Recovering input data for given CRC64 value of it

I've always wanted to do so, but everyone knows this is impossible for input buffers larger than 8 bytes. As my experiments show, it's still possible for tiny input buffers of data, which is constrained in some way.

The CRC64 value of 13-byte "undisciplined" string is known: `0x811265a32d6ac13a`. KLEE can find this 13-byte string, if it knows that each byte of input buffer is in *a..z* limits:

```
1  ...
2  
3  int main()
4  {
5  #define BUF_SIZE 13
6  
7          char buf[BUF_SIZE];
8  
9          klee_make_symbolic(buf, sizeof buf, "buf");
10 
11         for (int i=0; i<BUF_SIZE; i++)
12                 klee_assume (buf[i]>='a' && buf[i]<='z');
13 
14         klee_assume (crc64 (0, buf, BUF_SIZE)==0x811265a32d6ac13a);
15 }
```

```
% clang -emit-llvm -c -g -O0 -Xclang -disable-O0-optnone -I /tmp/klee_src/include/
    klee/ klee_CRC64_undisciplined.c
...

% time klee --libc=uclibc --optimize klee_CRC64_undisciplined.bc
...
real    0m13.948s
user    0m13.829s
...

% ls klee-last/*
klee-last/assembly.ll  klee-last/messages.txt  klee-last/run.stats            klee-
    last/test000001.ktest
klee-last/info         klee-last/run.istats    klee-last/solver-queries.smt2  klee-
    last/warnings.txt

% ktest-tool klee-last/test000001.ktest | grep data
object 0: data: b'undisciplined'
```

Still, it's no magic: if to remove condition at line 30 (i.e., if to relax constraint), KLEE will produce some other string, which will be still correct for the CRC64 value given.

It works, because 13 Latin characters in *a..z* limits contain ≈ 61 bits: $log_2 26^{13} \approx 61$, which is even smaller value than 64. In other words, the final CRC64 value holds enough bits to recover ≈ 61 bits of input.

The input buffer can be even bigger, if each byte of it will be in even tighter constraints (decimal digits, binary digits, etc).

13.1.4 Finding string that results in CRC64(string)==0

I always was wondering, how to find a string so that result of CRC64 operation will be zero. Again, this string is constrained: each character is in *a..z* range.

```
...

int main()
{
//#define BUF_SIZE 13 // fail
#define BUF_SIZE 14

        char buf[BUF_SIZE];
        klee_make_symbolic(buf, sizeof buf, "buf");

        for (int i=0; i<BUF_SIZE; i++)
                klee_assume (buf[i]>='a' && buf[i]<='z');

        klee_assume (crc64 (0, buf, BUF_SIZE)==0);
}
```

```
% ktest-tool klee-last/test000001.ktest | grep data
object 0: data: b'adnwsskbroggni'
```

So `CRC64("adnwsskbroggni")==0`, cool.

Buf shorter string (13 character) isn't possible, alas.

Every tool has its limitations, so is KLEE. I couldn't find such string with characters in *0..9* range.

13.2 Now CBMC

Here I reimplement KLEE examples you already saw above. All timings are when KLEE 3.0 running on AMD Ryzen 5 3600.

I'm not using `klee_assume()` calls here, only standard `assert()`'s.

```
#include <stdint.h>
#include <stdbool.h>

// CRC-64-ECMA in https://en.wikipedia.org/wiki/Cyclic_redundancy_check
uint64_t crc64(uint64_t crc, unsigned char *buf, int len)
{
        int k;

        crc = ~crc;
        while (len--)
        {
                crc ^= *buf++;
                for (k = 0; k < 8; k++)
                        crc = crc & 1 ? (crc >> 1) ^ 0xC96C5795D7870F42 : crc >> 1;
        }
        return ~crc;
}

#define SIZE 13

void main()
{
        char str[SIZE];

        int i=0;
        for (i=0; i<SIZE; i++)
        {
                if (str[i]<'a')
                        return;
                if (str[i]>'z')
                        return;
        };

        if (crc64(0, &str[0], SIZE)==0x811265a32d6ac13a)
                assert (0);
};
```

Listing 13.2: Command to run

```
% cbmc --trace --function main --z3 CBMC_crc64_find.c
```

```
** Results:
CBMC_crc64_find.c function main
[main.assertion.1] line 35 assertion 0: FAILURE

Trace for main.assertion.1:

State 21 file CBMC_crc64_find.c function main line 23 thread 0
----------------------------------------------------
  str={ 'u', 'n', 'd', 'i', 's', 'c', 'i', 'p', 'l', 'i', 'n', 'e', 'd' } ({
      01110101, 01101110, 01100100, 01101001, 01110011, 01100011, 01101001, 01110000,
      01101100, 01101001, 01101110, 01100101, 01100100 })

...

Violated property:
  file CBMC_crc64_find.c function main line 35 thread 0
  assertion 0
  (__CPROVER_bool)0



** 1 of 1 failed (2 iterations)
VERIFICATION FAILED
```

This result coincides with KLEE's result.

Now a string for CRC64(...)==0:

```
...

#define SIZE 14

void main()
{
        char str[SIZE];

        int i=0;
        for (i=0; i<SIZE; i++)
        {
                if (str[i]<'a')
                        return;
                if (str[i]>'z')
                        return;
        };

        if (crc64(0, &str[0], SIZE)==0)
                assert (0);
};
```

```
** Results:
CBMC_crc64_find_2.c function main
[main.assertion.1] line 36 assertion 0: FAILURE

Trace for main.assertion.1:

State 11 file CBMC_crc64_find_2.c function main line 24 thread 0
----------------------------------------------------
  str={ 'a', 'd', 'n', 'w', 's', 's', 'k', 'b', 'r', 'o', 'g', 'g', 'n', 'i' } ({
      01100001, 01100100, 01101110, 01110111, 01110011, 01110011, 01101011, 01100010,
      01110010, 01101111, 01100111, 01100111, 01101110, 01101001 })
```

Now a 16-byte string where each character is a nibble? Or, in other word, 16-nibble hexadecimal string? Or, in other words, 8-byte hexadecimal string?

```
...

#define SIZE 16

int valid_char(char c)
{
        if (c>='0' && c<='9')
                return 1;
        if (c>='a' && c<='f')
                return 1;
        return 0;
};

void main()
{
        char str[SIZE];

        int i=0;
        for (i=0; i<SIZE; i++)
        {
                if (valid_char(str[i])==0)
                        return;
        };

        if (crc64(0, &str[0], SIZE)==0)
                assert (0);
};
```

```
** Results:
CBMC_crc64_find_3.c function main
[main.assertion.1] line 45 assertion 0: FAILURE

Trace for main.assertion.1:

State 11 file CBMC_crc64_find_3.c function main line 35 thread 0
----------------------------------------------------
  str={ '8', '4', '9', '3', 'a', '5', '6', '4', '8', '6', '0', '0', 'c', '4', '9',
      '5' } ({ 00111000, 00110100, 00111001, 00110011, 01100001, 00110101, 00110110,
      00110100, 00111000, 00110110, 00110000, 00110000, 01100011, 00110100, 00111001,
      00110101 })

...

Violated property:
  file CBMC_crc64_find_3.c function main line 45 thread 0
  assertion 0
  0 != 0



** 1 of 1 failed (2 iterations)
VERIFICATION FAILED
['cbmc', '--trace', '--function', 'main', '--z3', 'CBMC_crc64_find_3.c']
['/home/i/dotfiles/bin/my_time.py', 'cbmc', '--trace', '--function', 'main', '--z3',
    'CBMC_crc64_find_3.c']
seconds:  510
or:  8m30s
```

Yes, CRC64("8493a5648600c495")==0, and this is very predictable: we just need to find a 64-bit input for CRC64 so that it's output would be zero. No magic here.

Again, CBMC can't find a string consisting only of *0..9* characters, like KLEE.

Thanks to Martin Nyx Brain[4], for help.

13.3 Factorize GF(2)/CRC polynomials

GF(2)/CRC polynomials, like usual numbers, can also be factored, because a polynomial can be a product of two other polynomial (or not).

Some people say that good CRC polynomial should be irreducible (i.e., cannot be factored), some other say that this is not a requirement. I've checked several CRC-16 and CRC-32 polynomials from the Wikipedia article.

The multiplier is constructed in the same manner, as I did it earlier for integer factorization using SAT. Factors are not prime integers, but prime polynomials.

Another important thing to notice is that replacing XOR with addition will make this script factor integers, because addition in GF(2) is XOR.

Also, can be used for tests, online GF(2) polynomials factorization: http://www.ee.unb.ca/cgi-bin/tervo/factor.pl?binary=101.

Listing 13.3: For Z3

```
#!/usr/bin/env python3

import operator, functools
from z3 import *
import my_utils

"""
rows with dots are partial products:

     aaaa
b ___....
b __...._
b _....__
b ....___

width = OUTPUT_SIZE (INPUT_SIZE*2-1)
height = INPUT_SIZE

"""

def factor(poly):
    INPUT_SIZE=size=my_utils.ceil_binlog(poly)
    OUTPUT_SIZE=INPUT_SIZE*2

    a=BitVec('a', INPUT_SIZE)
    b=BitVec('b', INPUT_SIZE)

    # partial products
    p=[BitVec('p_%d' % i, OUTPUT_SIZE) for i in range(INPUT_SIZE)]

    s=Solver()

    for i in range(INPUT_SIZE):
        # if there is a bit in b[], assign shifted a[] padded with zeroes at left/
            right
        # if there is no bit in b[], let p[] be zero
```

[4] https://github.com/diffblue/cbmc/issues/5099

```
        # Concat() is for glueling together bitvectors (of different widths)
        # BitVecVal() is constant of specific width

        if i==0:
            s.add(p[i] == If((b>>i)&1==1, Concat(BitVecVal(0, OUTPUT_SIZE-i-
               INPUT_SIZE), a), 0))
        else:
            s.add(p[i] == If((b>>i)&1==1, Concat(BitVecVal(0, OUTPUT_SIZE-i-
               INPUT_SIZE), a, BitVecVal(0, i)), 0))

    # form expression like s.add(p[0] ^ p[1] ^ ... ^ p[OUTPUT_SIZE-1] == poly)
    # replace operator.xor to operator.add to factorize numbers:
    s.add(functools.reduce (operator.xor, p)==poly)

    # we are not interesting in outputs like these:
    s.add(a!=1)
    s.add(b!=1)

    if s.check()==unsat:
        print ("prime factor: 0x%x or %s" % (poly, my_utils.poly_to_str(poly)))
        return

    m=s.model()
    #print ("sat, a=0x%x, b=0x%x" % (m[a].as_long(), m[b].as_long()))
    # factor results recursively:
    factor(m[a].as_long())
    factor(m[b].as_long())

poly=int(sys.argv[1], 16)

print ("starting. poly=0x%x or %s" % (poly, my_utils.poly_to_str(poly)))
factor(poly)
```

See also almost the same solution for my SAT library: https://smt.st/current_tree/CRC/factor/gf2_factor_SAT.py

Listing 13.4: Test

```
#!/usr/bin/env bash

gf2_factor_test ()
{
    echo "* poly $1"
    echo "** running for SAT"
    ./gf2_factor_SAT.py $1
    echo "** running for Z3"
    ./gf2_factor_z3.py  $1
}

# MSB is always 1 in CRC polynomials, and it's omitted
# but we add it here (leading 1 bit):

gf2_factor_test 0x5 # reducible
gf2_factor_test 0x7 # irreducible

# Both irreducible [from Colbourn, Dinitz - Handbook of Combinatorial Designs (2ed,
   2007), p.809]
gf2_factor_test 0x409 # 10000001001
gf2_factor_test 0x40f # 10000001111

gf2_factor_test 0x11021 # CRC-16-CCITT, reducible
gf2_factor_test 0x18005 # CRC-16-IBM, reducible
```

```
gf2_factor_test 0x1C867 # CRC-16-CDMA2000, irreducible

gf2_factor_test 0x104c11db7 # CRC-32, irreducible
gf2_factor_test 0x11EDC6F41 # CRC-32C (Castagnoli), CRC32 x86 instruction, reducible
gf2_factor_test 0x132583499 # CRC-32K2 (Koopman {1,1,30}), reducible
gf2_factor_test 0x1741B8CD7 # CRC-32K (Koopman {1,3,28}), reducible
gf2_factor_test 0x1814141AB # CRC-32Q, reducible

gf2_factor_test 0x142F0E1EBA9EA3693 # CRC-64-ECMA - reducible
#0x1000000000000001B # CRC-64-ISO. timeout

#0x100000000000000000000000000000087 # AES-GCM (not CRC)
```

Listing 13.5: Results

```
* poly 0x5
** running for SAT
starting. poly=0x5 or a^2 + 1
prime factor: 0x3 or a + 1
prime factor: 0x3 or a + 1
** running for Z3
starting. poly=0x5 or a^2 + 1
prime factor: 0x3 or a + 1
prime factor: 0x3 or a + 1
* poly 0x7
** running for SAT
starting. poly=0x7 or a^2 + a + 1
prime factor: 0x7 or a^2 + a + 1
** running for Z3
starting. poly=0x7 or a^2 + a + 1
prime factor: 0x7 or a^2 + a + 1
* poly 0x409
** running for SAT
starting. poly=0x409 or a^10 + a^3 + 1
prime factor: 0x409 or a^10 + a^3 + 1
** running for Z3
starting. poly=0x409 or a^10 + a^3 + 1
prime factor: 0x409 or a^10 + a^3 + 1
* poly 0x40f
** running for SAT
starting. poly=0x40f or a^10 + a^3 + a^2 + a + 1
prime factor: 0x40f or a^10 + a^3 + a^2 + a + 1
** running for Z3
starting. poly=0x40f or a^10 + a^3 + a^2 + a + 1
prime factor: 0x40f or a^10 + a^3 + a^2 + a + 1
* poly 0x11021
** running for SAT
starting. poly=0x11021 or a^16 + a^12 + a^5 + 1
prime factor: 0xf01f or a^15 + a^14 + a^13 + a^12 + a^4 + a^3 + a^2 + a + 1
prime factor: 0x3 or a + 1
** running for Z3
starting. poly=0x11021 or a^16 + a^12 + a^5 + 1
prime factor: 0xf01f or a^15 + a^14 + a^13 + a^12 + a^4 + a^3 + a^2 + a + 1
prime factor: 0x3 or a + 1
* poly 0x18005
** running for SAT
starting. poly=0x18005 or a^16 + a^15 + a^2 + 1
prime factor: 0x3 or a + 1
prime factor: 0x8003 or a^15 + a + 1
** running for Z3
starting. poly=0x18005 or a^16 + a^15 + a^2 + 1
```

```
prime factor: 0x8003 or a^15 + a + 1
prime factor: 0x3 or a + 1
* poly 0x1C867
** running for SAT
starting. poly=0x1c867 or a^16 + a^15 + a^14 + a^11 + a^6 + a^5 + a^2 + a + 1
prime factor: 0x1c867 or a^16 + a^15 + a^14 + a^11 + a^6 + a^5 + a^2 + a + 1
** running for Z3
starting. poly=0x1c867 or a^16 + a^15 + a^14 + a^11 + a^6 + a^5 + a^2 + a + 1
prime factor: 0x1c867 or a^16 + a^15 + a^14 + a^11 + a^6 + a^5 + a^2 + a + 1
* poly 0x104c11db7
** running for SAT
starting. poly=0x104c11db7 or a^32 + a^26 + a^23 + a^22 + a^16 + a^12 + a^11 + a^10 +
     a^8 + a^7 + a^5 + a^4 + a^2 + a + 1
prime factor: 0x104c11db7 or a^32 + a^26 + a^23 + a^22 + a^16 + a^12 + a^11 + a^10 +
    a^8 + a^7 + a^5 + a^4 + a^2 + a + 1
** running for Z3
starting. poly=0x104c11db7 or a^32 + a^26 + a^23 + a^22 + a^16 + a^12 + a^11 + a^10 +
     a^8 + a^7 + a^5 + a^4 + a^2 + a + 1
prime factor: 0x104c11db7 or a^32 + a^26 + a^23 + a^22 + a^16 + a^12 + a^11 + a^10 +
    a^8 + a^7 + a^5 + a^4 + a^2 + a + 1
* poly 0x11EDC6F41
** running for SAT
starting. poly=0x11edc6f41 or a^32 + a^28 + a^27 + a^26 + a^25 + a^23 + a^22 + a^20 +
     a^19 + a^18 + a^14 + a^13 + a^11 + a^10 + a^9 + a^8 + a^6 + 1
prime factor: 0x3 or a + 1
prime factor: 0xf5b4253f or a^31 + a^30 + a^29 + a^28 + a^26 + a^24 + a^23 + a^21 + a
    ^20 + a^18 + a^13 + a^10 + a^8 + a^5 + a^4 + a^3 + a^2 + a + 1
** running for Z3
starting. poly=0x11edc6f41 or a^32 + a^28 + a^27 + a^26 + a^25 + a^23 + a^22 + a^20 +
     a^19 + a^18 + a^14 + a^13 + a^11 + a^10 + a^9 + a^8 + a^6 + 1
prime factor: 0x3 or a + 1
prime factor: 0xf5b4253f or a^31 + a^30 + a^29 + a^28 + a^26 + a^24 + a^23 + a^21 + a
    ^20 + a^18 + a^13 + a^10 + a^8 + a^5 + a^4 + a^3 + a^2 + a + 1
* poly 0x132583499
** running for SAT
starting. poly=0x132583499 or a^32 + a^29 + a^28 + a^25 + a^22 + a^20 + a^19 + a^13 +
     a^12 + a^10 + a^7 + a^4 + a^3 + 1
prime factor: 0x3 or a + 1
prime factor: 0x5a12a42d or a^30 + a^28 + a^27 + a^25 + a^20 + a^17 + a^15 + a^13 + a
    ^10 + a^5 + a^3 + a^2 + 1
prime factor: 0x3 or a + 1
** running for Z3
starting. poly=0x132583499 or a^32 + a^29 + a^28 + a^25 + a^22 + a^20 + a^19 + a^13 +
     a^12 + a^10 + a^7 + a^4 + a^3 + 1
prime factor: 0x5a12a42d or a^30 + a^28 + a^27 + a^25 + a^20 + a^17 + a^15 + a^13 + a
    ^10 + a^5 + a^3 + a^2 + 1
prime factor: 0x3 or a + 1
prime factor: 0x3 or a + 1
* poly 0x1741B8CD7
** running for SAT
starting. poly=0x1741b8cd7 or a^32 + a^30 + a^29 + a^28 + a^26 + a^20 + a^19 + a^17 +
     a^16 + a^15 + a^11 + a^10 + a^7 + a^6 + a^4 + a^2 + a + 1
prime factor: 0xd or a^3 + a^2 + 1
prime factor: 0x10595341 or a^28 + a^22 + a^20 + a^19 + a^16 + a^14 + a^12 + a^9 + a
    ^8 + a^6 + 1
prime factor: 0x3 or a + 1
** running for Z3
starting. poly=0x1741b8cd7 or a^32 + a^30 + a^29 + a^28 + a^26 + a^20 + a^19 + a^17 +
     a^16 + a^15 + a^11 + a^10 + a^7 + a^6 + a^4 + a^2 + a + 1
prime factor: 0x10595341 or a^28 + a^22 + a^20 + a^19 + a^16 + a^14 + a^12 + a^9 + a
    ^8 + a^6 + 1
```

```
prime factor: 0xd or a^3 + a^2 + 1
prime factor: 0x3 or a + 1
* poly 0x1814141AB
** running for SAT
starting. poly=0x1814141ab or a^32 + a^31 + a^24 + a^22 + a^16 + a^14 + a^8 + a^7 + a
   ^5 + a^3 + a + 1
prime factor: 0x3 or a + 1
prime factor: 0x80c0c099 or a^31 + a^23 + a^22 + a^15 + a^14 + a^7 + a^4 + a^3 + 1
** running for Z3
starting. poly=0x1814141ab or a^32 + a^31 + a^24 + a^22 + a^16 + a^14 + a^8 + a^7 + a
   ^5 + a^3 + a + 1
prime factor: 0x3 or a + 1
prime factor: 0x80c0c099 or a^31 + a^23 + a^22 + a^15 + a^14 + a^7 + a^4 + a^3 + 1
* poly 0x142F0E1EBA9EA3693
** running for SAT
starting. poly=0x142f0e1eba9ea3693 or a^64 + a^62 + a^57 + a^55 + a^54 + a^53 + a^52
   + a^47 + a^46 + a^45 + a^40 + a^39 + a^38 + a^37 + a^35 + a^33 + a^32 + a^31 + a
   ^29 + a^27 + a^24 + a^23 + a^22 + a^21 + a^19 + a^17 + a^13 + a^12 + a^10 + a^9 +
   a^7 + a^4 + a + 1
prime factor: 0x3 or a + 1
prime factor: 0x3 or a + 1
prime factor: 0x25f39 or a^17 + a^14 + a^12 + a^11 + a^10 + a^9 + a^8 + a^5 + a^4 + a
   ^3 + 1
prime factor: 0x900b or a^15 + a^12 + a^3 + a + 1
prime factor: 0x8003 or a^15 + a + 1
prime factor: 0x8423 or a^15 + a^10 + a^5 + a + 1
** running for Z3
starting. poly=0x142f0e1eba9ea3693 or a^64 + a^62 + a^57 + a^55 + a^54 + a^53 + a^52
   + a^47 + a^46 + a^45 + a^40 + a^39 + a^38 + a^37 + a^35 + a^33 + a^32 + a^31 + a
   ^29 + a^27 + a^24 + a^23 + a^22 + a^21 + a^19 + a^17 + a^13 + a^12 + a^10 + a^9 +
   a^7 + a^4 + a + 1
prime factor: 0x8003 or a^15 + a + 1
prime factor: 0x900b or a^15 + a^12 + a^3 + a + 1
prime factor: 0x25f39 or a^17 + a^14 + a^12 + a^11 + a^10 + a^9 + a^8 + a^5 + a^4 + a
   ^3 + 1
prime factor: 0x8423 or a^15 + a^10 + a^5 + a + 1
prime factor: 0x3 or a + 1
prime factor: 0x3 or a + 1
```

13.3.1 SageMath

... but.

SageMath can do this way more quickly, and can factor 64-bit polynomials on which my solution stuck. Also, it can factor 128-bit polynomial used in AES-GCM, with no effort.

https://smt.st/current_tree/CRC/factor/sagemath/check_irreducible.html.

(SageMath notebooks I used: https://smt.st/current_tree/CRC/factor/sagemath/.)

Alas! This is a professional mathematical tool. It probably uses a specially tailored algorithm, which may be way more complex.

Yet again, I wrote my tool for SAT/SMT with no special knowledge, as a newbie. And it was fun and it helped me to learn Galois fields.

13.4 Getting CRC polynomial and other CRC generator parameters

Sometimes CRC implementations are incompatible with each other: polynomial and other parameters can be different. Aside of polynomial, initial state can be either 0 or -1, final value can be inverted or not, bits of the final value can be reflected (reversed) or not. Input byte can be reflected/reversed before processing as well. Trying all these parameters by hand to match with someone's else implementation can be a real pain. Also, you can bruteforce 32-bit polynomial, but 64-bit polynomials is too much.

Deducing all these parameters is surprisingly simple using Z3, just get two values for 01 byte and 02, or any other bytes.

```
#!/usr/bin/env python3

from z3 import *
import struct

# knobs:

# CRC-16 on https://www.lammertbies.nl/comm/info/crc-calculation.html
#width=16
#samples=["\x01", "\x02"]
#must_be=[0xC0C1, 0xC181]
#sample_len=1

# CRC-16 (Modbus) on https://www.lammertbies.nl/comm/info/crc-calculation.html
#width=16
#samples=["\x01", "\x02"]
#must_be=[0x807E, 0x813E]
#sample_len=1

# crc-32, popular parameters
width=32
samples=["\x01", "\x02"]
must_be=[0xA505DF1B, 0x3C0C8EA1]
sample_len=1

# crc64_jones.c (redis):
#width=64
#samples=["\x01", "\x02"]
#must_be=[0x7ad870c830358979, 0xf5b0e190606b12f2]
#sample_len=1

# crc64_xz.c:
#width=64
#samples=["\x01", "\x02"]
#must_be=[0xac83edcd67c06036, 0xeb299724cc279f02]
#sample_len=1

# http://www.unit-conversion.info/texttools/crc/
#width=32
#samples=["0","1"]
#must_be=[0xf4dbdf21, 0x83dcefb7]
#sample_len=1

# recipe-259177-1.py, CRC-64-ISO
#width=64
#samples=["\x01", "\x02"]
#must_be=[0x01B0000000000000, 0x0360000000000000]
#sample_len=1

# recipe-259177-1.py, CRC-64-ISO
# many solutions!
#width=64
#samples=["\x01"]
#must_be=[0x01B0000000000000]
#sample_len=1

# crc-32, popular parameters
# slower
#width=32
#samples=["12","ab"]
```

```
#must_be=[0x4F5344CD, 0x9E83486D]
#sample_len=2

# crc64 ecma 182
#width=64
#samples=["\x01", "\x02"]
#must_be=[0x42f0e1eba9ea3693, 0x85e1c3d753d46d26]
#sample_len=1

# not z3 function (see underscore)
def LShR_(x, cnt):
    return x>>cnt

def bitrev8(x):
    x = LShR_(x, 4) | (x << 4)
    x = LShR_(x & 0xCC, 2) | ((x & 0x33) << 2)
    x = LShR_(x & 0xAA, 1) | ((x & 0x55) << 1)
    return x

# these "unoptimized" versions are constructed like a Russian doll...

def bitrev16_unoptimized(x):
    return (bitrev8(x & 0xff) << 8) | (bitrev8(LShR_(x, 8)))

def bitrev32_unoptimized(x):
    return (bitrev16_unoptimized(x & 0xffff) << 16) | (bitrev16_unoptimized(LShR_(x,
       16)))

def bitrev64_unoptimized(x):
    # both versions must work:
    return (bitrev32_unoptimized(x & 0xffffffff) << 32) | bitrev32_unoptimized(LShR_(
       x, 32))
    #return (bitrev32(x & 0xffffffff) << 32) | bitrev32(LShR(x, 32))

def bitrev(width, val):
    if width==64:
        return bitrev64_unoptimized(val)
    if width==32:
        return bitrev32_unoptimized(val)
    if width==16:
        return bitrev16_unoptimized(val)
    raise AssertionError

mask=2**width-1
poly=BitVec('poly', width)

# states[sample][0][8] is an initial state
# ...
# states[sample][i][0] is a state where it was already XORed with input bit
# states[sample][i][1] ... where the 1th shift/XOR operation has been done
# states[sample][i][8] ... where the 8th shift/XOR operation has been done
# ...
# states[sample][sample_len][8] - final state

states=[[[BitVec('state_%d_%d_%d' % (sample, i, bit), width) for bit in range(8+1)]
   for i in range(sample_len+1)] for sample in range(len(samples))]
s=Solver()

reflect_in=Bool("reflect_in")

def invert(val):
    return ~val & mask

for sample in range(len(samples)):
    # initial state can be either zero or -1:
    s.add(Or(states[sample][0][8]==mask, states[sample][0][8]==0))

    # implement basic CRC algorithm
    for i in range(sample_len):
        x=BitVecVal(ord(samples[sample][i]), width) # ReflectIn True
        y=BitVecVal(bitrev8(ord(samples[sample][i])), width) # ReflectIn False
        s.add(states[sample][i+1][0] == (states[sample][i][8] ^ If(reflect_in, x, y))
            )

        for bit in range(8):
            # LShR() is logical shift, while >> is arithmetical shift, we use the
                first:
            s.add(states[sample][i+1][bit+1] == LShR(states[sample][i+1][bit],1) ^ If
                (states[sample][i+1][bit]&1==1, poly, 0))

    # final state must be equal to one of these:
    s.add(Or(
        states[sample][sample_len][8]==must_be[sample],
        states[sample][sample_len][8]==invert(must_be[sample]),
        states[sample][sample_len][8]==bitrev(width, must_be[sample]),
        states[sample][sample_len][8]==invert(bitrev(width, must_be[sample]))))

# get all possible results:
results=[]
while True:
    if s.check() == sat:
        m = s.model()
        # what final state was?
        if m[states[0][sample_len][8]].as_long()==must_be[0]:
            outparams="XORout=0, ReflectOut=True"
        elif invert(m[states[0][sample_len][8]].as_long())==must_be[0]:
            outparams="XORout=-1, ReflectOut=True"
        elif m[states[0][sample_len][8]].as_long()==bitrev(width, must_be[0]):
            outparams="XORout==0, ReflectOut=False"
        elif invert(m[states[0][sample_len][8]].as_long())==bitrev(width, must_be[0])
            :
            outparams="XORout=-1, ReflectOut=False"
        else:
            raise AssertionError

        print ("poly=0x%x, init=0x%x, ReflectIn=%s, %s" % (m[poly].as_long(), m[
            states[0][0][8]].as_long(), str(m[reflect_in]), outparams))

        results.append(m)
        block = []
        for d in m:
            c=d()
            block.append(c != m[d])
        s.add(Or(block))
    else:
        print ("total results", len(results))
        break
```

This is for CRC-16 (Modbus):

```
poly=0xa001, init=0xffff, ReflectIn=True, XORout=0, ReflectOut=True
```

Sometimes, we have no enough information, but still can get something. This is for CRC-16:

```
poly=0x96bc, init=0x0, ReflectIn=True, XORout=-1, ReflectOut=True
poly=0x7cfc, init=0x0, ReflectIn=False, XORout=-1, ReflectOut=False
poly=0x5814, init=0x0, ReflectIn=True, XORout=-1, ReflectOut=True
poly=0x6182, init=0x0, ReflectIn=True, XORout=-1, ReflectOut=True
poly=0xa287, init=0x0, ReflectIn=True, XORout=-1, ReflectOut=True
poly=0x726b, init=0x0, ReflectIn=True, XORout=-1, ReflectOut=True
poly=0x41c1, init=0x0, ReflectIn=True, XORout=-1, ReflectOut=True
poly=0xe83d, init=0x0, ReflectIn=True, XORout=-1, ReflectOut=True
poly=0xa001, init=0x0, ReflectIn=True, XORout=0, ReflectOut=True
total results 9
```

One of these results is correct (last).

We can get something even if we have only one result for one input byte:

```
# recipe-259177-1.py, CRC-64-ISO
# many solutions!
width=64
samples=["\x01"]
must_be=[0x01B0000000000000]
sample_len=1
```

```
poly=0xd800000000000000, init=0x0, ReflectIn=True, XORout=0, ReflectOut=True
poly=0xfe4fffffffffffff, init=0x0, ReflectIn=False, XORout=-1, ReflectOut=True
poly=0xfffffffffffff27f, init=0x0, ReflectIn=False, XORout=-1, ReflectOut=False
poly=0xd80, init=0x0, ReflectIn=False, XORout==0, ReflectOut=False
poly=0x1b0000000000000, init=0x0, ReflectIn=False, XORout=0, ReflectOut=True
poly=0xa7ffffffffffffbf, init=0xffffffffffffffff, ReflectIn=False, XORout=0,
   ReflectOut=True
poly=0x6c000, init=0x0, ReflectIn=True, XORout==0, ReflectOut=False
poly=0xb40b40b40b4816, init=0xffffffffffffffff, ReflectIn=True, XORout==0, ReflectOut
   =False
poly=0xe73cf3cf3cf3cf3c, init=0xffffffffffffffff, ReflectIn=False, XORout=-1,
   ReflectOut=True
poly=0x53ffffffffffffff, init=0xffffffffffffffff, ReflectIn=True, XORout=0,
   ReflectOut=True
poly=0x7ffffffffff93fbf, init=0xffffffffffffffff, ReflectIn=False, XORout==0,
   ReflectOut=False
poly=0xcc2055dc9e9bc60f, init=0xffffffffffffffff, ReflectIn=False, XORout=-1,
   ReflectOut=True
poly=0x3ffffffffffc9fff, init=0xffffffffffffffff, ReflectIn=True, XORout==0,
   ReflectOut=False
poly=0x9fd82e25acc7fcf3, init=0xffffffffffffffff, ReflectIn=False, XORout=-1,
   ReflectOut=False
poly=0x367a57106cf7678, init=0xffffffffffffffff, ReflectIn=False, XORout==0,
   ReflectOut=False
poly=0x92492492492e924, init=0xffffffffffffffff, ReflectIn=True, XORout==0,
   ReflectOut=False
poly=0x1d24924924924924, init=0xffffffffffffffff, ReflectIn=True, XORout=0,
   ReflectOut=True
poly=0x31ba, init=0x0, ReflectIn=True, XORout==0, ReflectOut=False
poly=0x153225b1d0d61af, init=0xffffffffffffffff, ReflectIn=False, XORout==0,
   ReflectOut=False
poly=0x461861861861861, init=0xffffffffffffffff, ReflectIn=True, XORout=0, ReflectOut
   =True
poly=0x30c30c30c30f861, init=0xffffffffffffffff, ReflectIn=True, XORout==0,
   ReflectOut=False
total results 21
```

(The first result is correct.)

The files: https://smt.st/current_tree/CRC/cracker.

The shortcoming: longer samples slows down everything significantly. I had luck with samples up to 4 bytes, but not larger.

Further reading I've found interesting/helpful:

- http://www.cosc.canterbury.ac.nz/greg.ewing/essays/CRC-Reverse-Engineering.html
- http://reveng.sourceforge.net/crc-catalogue/1-15.htm
- http://reveng.sourceforge.net/crc-catalogue/16.htm
- http://reveng.sourceforge.net/crc-catalogue/17plus.htm

13.5 Finding (good) CRC polynomial

Finding good CRC polynomial is tricky, and my results can't compete with other tested popular CRC polynomial. Nevertheless, it was fun to use Z3 to find them.

I just generate 32 random samples, all has size between 1 and 32 bytes. Then I flip 1..3 random bits and I add a constraint: CRC hash of the sample and hash of the modified sample (with 1..3 bits flipped) must differ.

```
#!/usr/bin/env python3

from z3 import *
import copy, random

width=32

poly=BitVec('poly', width)

s=Solver()

no_call=0

def CRC(_input, poly):
    # make each variable name unique
    # no_call (number of call) increments at each call to CRC() function
    global no_call
    states=[[BitVec('state_%d_%d_%d' % (no_call, i, bit), width) for bit in range
        (8+1)] for i in range(len(_input)+1)]
    no_call=no_call+1
    # initial state is always 0:
    s.add(states[0][8]==0)

    for i in range(len(_input)):
        s.add(states[i+1][0] == states[i][8] ^ _input[i])

        for bit in range(8):
            s.add(states[i+1][bit+1] == LShR(states[i+1][bit],1) ^ If(states[i+1][bit
                ]&1==1, poly, 0))

    return states[len(_input)][8]

# generate 32 random samples:
for i in range(32):
    print ("pair",i)
    # each sample has random size 1..32
    buf1=bytearray(os.urandom(random.randrange(32)+1))
    buf2=copy.deepcopy(buf1)
    # flip 1, 2 or 3 random bits in second sample:
    for bits in range(1,random.randrange(3)+2):
        # get random position and bit to flip:
        pos=random.randrange(0, len(buf2))
        to_flip=1<<random.randrange(8)
        print ("  pos=", pos, "bit=",to_flip)
        # flip random bit at random position:
        buf2[pos]=buf2[pos]^to_flip

    # original sample and sample with 1..3 random bits flipped.
    # their hashes must be different:
    s.add(CRC(buf1, poly)!=CRC(buf2, poly))

# get all possible results:
results=[]
while True:
    if s.check() == sat:
        m = s.model()
        print ("poly=0x%x" % (m[poly].as_long()))
        results.append(m)
        block = []
        for d in m:
            c=d()
            block.append(c != m[d])
        s.add(Or(block))
    else:
        print ("total results", len(results))
        break
```

Several polynomials for CRC8:

```
poly=0xf9
poly=0x50
poly=0x90
...
```

... for CRC16:

```
poly=0xf7af
poly=0x368
poly=0x268
poly=0x228
...
```

... for CRC32:

```
poly=0x1683a5ab
poly=0x78553eda
poly=0x7a153eda
poly=0x7b353eda
...
```

... for CRC64:

```
poly=0x8000000000000006
poly=0x926b19b536a62f10
poly=0x4a7bb0a7da78a370
poly=0xbbc781e7e83dabf0
...
```

Problem: at least this one. CRC must be able to detect errors in very long buffers, up to 2^{32} for CRC32. We can't feed that huge buffers to SMT solver. I had success only with samples up to ≈ 32 bytes.

13.6 In comparison with other hashing algorithms

Things are that easy for some other hashing algorithms like *Fletcher checksum*, but not for cryptographically secure ones (like MD5, SHA1, etc), they are protected from such simple cryptoanalysis. See also: 19.

Chapter 14

MaxSAT/MaxSMT

14.1 Making smallest possible test suite using Z3

I once worked on rewriting large piece of code into pure C, and there were a tests, several thousands. Testing process was painfully slow, so I thought if the test suite can be minimized somehow.

(This is also called *over-testing* or *test duplication*.)

What we can do is to run each test and get code coverage (information about which lines of code was executed and which are not). Then the task is to make such test suite, where coverage is maximum, and number of tests is minimal.

In fact, this is *set cover problem* (also known as *hitting set problem*). While simpler algorithms exist (see Wikipedia[1]), it is also possible to solve with SMT-solver.

First, I took LZSS[2] compression/decompression code [3] for the example, from Apple sources. Such routines are not easy to test. Here is my version of it: `https://smt.st/current_tree/MaxSxT/min_test_Z3/compression.c`. I've added random generation of input data to be compressed. Random generation is dependent of some kind of input seed. Standard `srand()`/`rand()` are not recommended to be used, but for such simple task as ours, it's OK. I'll generate[4] 1000 tests with 0..999 seeds, that would produce random data to be compressed/decompressed/checked.

After the compression/decompression routine has finished its work, GNU gcov utility is executed, which produces result like this:

```
...
     3395:  189:        for (i = 1; i < F; i++) {
     3395:  190:            if ((cmp = key[i] - sp->text_buf[p + i]) != 0)
     2565:  191:                break;
        -:  192:        }
     2565:  193:        if (i > sp->match_length) {
     1291:  194:            sp->match_position = p;
     1291:  195:            if ((sp->match_length = i) >= F)
    #####:  196:                break;
        -:  197:        }
     2565:  198:    }
    #####:  199:    sp->parent[r] = sp->parent[p];
    #####:  200:    sp->lchild[r] = sp->lchild[p];
    #####:  201:    sp->rchild[r] = sp->rchild[p];
    #####:  202:    sp->parent[sp->lchild[p]] = r;
    #####:  203:    sp->parent[sp->rchild[p]] = r;
    #####:  204:    if (sp->rchild[sp->parent[p]] == p)
    #####:  205:        sp->rchild[sp->parent[p]] = r;
...
```

A leftmost number is an execution count for each line. ##### means the line of code hasn't been executed at all. The second column is a line number.

Now the Z3Py script, which will parse all these 1000 gcov results and produce minimal *hitting set*:

```
#!/usr/bin/env python3
```

[1] https://en.wikipedia.org/wiki/Set_cover_problem

[2] Lempel–Ziv–Storer–Szymanski

[3] https://github.com/opensource-apple/kext_tools/blob/master/compression.c

[4] https://smt.st/current_tree/MaxSxT/min_test_Z3/gen_gcov_tests.sh

```
import re, sys
from z3 import *

TOTAL_TESTS=1000

# read gcov result and return list of lines executed:
def process_file (fname):
    lines=[]
    f=open(fname,"r")

    while True:
        l=f.readline().rstrip()
        m = re.search('^ *([0-9]+):  ([0-9]+):.*$', l)
        if m!=None:
            lines.append(int(m.group(2)))
        if len(l)==0:
            break

    f.close()
    return lines

# k=test number; v=list of lines executed
stat={}
for test in range(TOTAL_TESTS):
    stat[test]=process_file("compression.c.gcov."+str(test))

# that will be a list of all lines in all tests:
all_lines=set()
# k=line, v=list of tests, which trigger that line:
tests_for_line={}

for test in stat:
    all_lines|=set(stat[test])
    for line in stat[test]:
        tests_for_line[line]=tests_for_line.get(line, []) + [test]

# int variable for each test:
tests=[Int('test_%d' % (t)) for t in range(TOTAL_TESTS)]

# this is optimization problem, so Optimize() instead of Solver():
opt = Optimize()

# each test variable is either 0 (absent) or 1 (present):
for t in tests:
    opt.add(Or(t==0, t==1))

# we know which tests can trigger each line
# so we enumerate all tests when preparing expression for each line
# we form expression like "test_1==1 OR test_2==1 OR ..." for each line:
for line in list(all_lines):
    expressions=[tests[s]==1 for s in tests_for_line[line]]
    # expression is a list which unfolds as list of arguments into Z3's Or() function
        (see asterisk)
    # that results in big expression like "Or(test_1==1, test_2==1, ...)"
    # the expression is then added as a constraint:
    opt.add(Or(*expressions))

# we need to find a such solution, where minimal number of all "test_X" variables
   will have 1
# "*tests" unfolds to a list of arguments: [test_1, test_2, test_3,...]
# "Sum(*tests)" unfolds to the following expression: "Sum(test_1, test_2, ...)"
# the sum of all "test_X" variables should be as minimal as possible:
h=opt.minimize(Sum(*tests))

print (opt.check())
m=opt.model()

# print all variables set to 1:
for t in tests:
    if m[t].as_long()==1:
        print (t)
```

And what it produces (~19s on my old Intel Quad-Core Xeon E3-1220 3.10GHz):

```
% time python set_cover.py
sat
test_7
test_48
test_134
python set_cover.py  18.95s user 0.03s system 99% cpu 18.988 total
```

We need just these 3 tests to execute (almost) all lines in the code: looks impressive, given the fact, that it would be notoriously hard to pick these tests by hand! The result can be checked easily, again, using gcov utility.

This is sometimes also called MaxSAT/MaxSxT — the problem is to find solution, but the solution where some variable/expression is maximal as possible, or minimal as possible.

Also, the code gives incorrect results on Z3 4.4.1, but working correctly on Z3 4.5.0 (so please upgrade). This is relatively fresh feature in Z3, so probably it was not stable in previous versions?

The files: https://smt.st/current_tree/MaxSxT/min_test_Z3.

Further reading: https://en.wikipedia.org/wiki/Set_cover_problem, https://en.wikipedia.org/wiki/Maximum_satisfiability_problem, https://en.wikipedia.org/wiki/Optimization_problem.

14.2 Making smallest possible test suite using OpenWBO

My previous example (14.1) was made up. And now I can do it better.

Once upon a time, I wrote my own x86/x64 disassembler. And a code like that is fragile and prone to bugs — any unnoticed typo can ruin a hour or two. How to test it? Just to be sure, I put as much to tests as possible. I enumerated all possible 2 bytes opcodes, etc. At some point, there were 12531 tests, like:

```
...
disas_test1(Fuzzy_True, (const byte*)"\xF3\x0F\x58\xF1", 3, "ADDSS XMM6, XMM1");
disas_test1(Fuzzy_True, (const byte*)"\xF3\x0F\x59\xC0", 3, "MULSS XMM0, XMM0");
disas_test1(Fuzzy_True, (const byte*)"\xF3\x0F\x5C\xD7", 3, "SUBSS XMM2, XMM7");
disas_test1(Fuzzy_True, (const byte*)"\xF3\x0F\x5E\xF8", 3, "DIVSS XMM7, XMM0");
disas_test1(Fuzzy_True, (const byte*)"\xF3\x42\x0F\x11\x74\x1D\x98", 3, "MOVSS [RBP+
    R11-68h], XMM6");
disas_test1(Fuzzy_True, (const byte*)"\xF3\x48\x0F\x2A\xC0", 3, "CVTSI2SS XMM0, RAX")
    ;
disas_test1(Fuzzy_True, (const byte*)"\xF3\x48\xAB", 3, "REP STOSQ");
disas_test1(Fuzzy_True, (const byte*)"\xF7\xED", 0x300, "IMUL EBP");
disas_test2_1op(Fuzzy_True, (const byte*)"\xFF\xD0", 0x300, "CALL RAX", 64);
disas_test2_2op(Fuzzy_True, (const byte*)"\x00\x4C\x8B\x94", 0x300, "ADD [RBX+RCX*4-6
    ch], CL", 8, 8);
disas_test2_2op(Fuzzy_True, (const byte*)"\x0D\x01\xFF\xFF\xFF", 0x48DB13, "OR EAX, 0
    FFFFFF01h", 32, 32);
disas_test2_2op(Fuzzy_True, (const byte*)"\x0F\xB7\x0D\x4F\x02\x5B\x06", 0x436E96, "
    MOVZX ECX, [69e70ech]", 32, 16);
disas_test2_2op(Fuzzy_True, (const byte*)"\x0F\xB7\x0D\xDE\x01\x5B\x06", 0x436F07, "
    MOVZX ECX, [69e70ech]", 32, 16);
...
```

Basically, these are pairs: opcode + disassembled output.

I modified my test code so that a test number can be set in arguments, and then prepended this to each test:

```
...
if (test_all || line==__LINE__) disas_test1(Fuzzy_True, (const byte*)"\xF3\x0F\x58\
    xF1", 3, "ADDSS XMM6, XMM1");
if (test_all || line==__LINE__) disas_test1(Fuzzy_True, (const byte*)"\xF3\x0F\x59\
    xC0", 3, "MULSS XMM0, XMM0");
if (test_all || line==__LINE__) disas_test1(Fuzzy_True, (const byte*)"\xF3\x0F\x5C\
    xD7", 3, "SUBSS XMM2, XMM7");
if (test_all || line==__LINE__) disas_test1(Fuzzy_True, (const byte*)"\xF3\x0F\x5E\
    xF8", 3, "DIVSS XMM7, XMM0");
if (test_all || line==__LINE__) disas_test1(Fuzzy_True, (const byte*)"\xF3\x42\x0F\
    x11\x74\x1D\x98", 3, "MOVSS [RBP+R11-68h], XMM6");
if (test_all || line==__LINE__) disas_test1(Fuzzy_True, (const byte*)"\xF3\x48\x0F\
    x2A\xC0", 3, "CVTSI2SS XMM0, RAX");
if (test_all || line==__LINE__) disas_test1(Fuzzy_True, (const byte*)"\xF3\x48\xAB",
    3, "REP STOSQ");
if (test_all || line==__LINE__) disas_test1(Fuzzy_True, (const byte*)"\xF7\xED", 0
    x300, "IMUL EBP");
if (test_all || line==__LINE__) disas_test2_1op(Fuzzy_True, (const byte*)"\xFF\xD0",
    0x300, "CALL RAX", 64);
if (test_all || line==__LINE__) disas_test2_2op(Fuzzy_True, (const byte*)"\x00\x4C\
    x8B\x94", 0x300, "ADD [RBX+RCX*4-6ch], CL", 8, 8);
...
```

Also any disassembler, including mine, has a *mega-table* or *mother-table* or *master-table* that contains all opcodes, flags, instruction names, etc.

And I added a debugging statement: if a table entry is loaded, its number is being printed. And you'll see why.

I'm compiling the code so that it will generate GCOV-statistics:

```
gcc -fprofile-arcs -ftest-coverage -g -DX86_DISASM_PRINT_INS_TBL_ENTRY
    x86_disasm_tests.c x86_disas.c x86_register.c -I../octothorpe ../octothorpe/
    octothorpe.a
```

And I run it 12531 times. A number in 1..12531 range is passed in arguments:

```
#!/bin/bash
for i in $(seq 1 12532);
do

        echo $i
        rm *gcda*
        tbl_entry=$(./a.out $i | tail -1)
        echo $tbl_entry
        gcov x86_disas
        mv x86_disas.c.gcov gcovs/$i.$tbl_entry
done
```

12531 files have been created. Filename consists of number1.number2, where number1 is a test number (or line number in tests.h) and number2 is a table entry loaded during testing. The contents is a typical GCOV's output:

```
...
        1:  651:                if (IS_SET (p->new_flags,
            F_REG32_IS_LOWEST_PART_OF_1ST_BYTE))
    #####:  652:                        mask=0xF8;
        1:  653:                if (IS_SET (p->new_flags,
            F_REG64_IS_LOWEST_PART_OF_1ST_BYTE))
    #####:  654:                        mask=0xF8;
        -:  655:
        1:  656:                if ((opc & mask) != ins_tbl[p->tbl_p].opc)
        -:  657:                {
    #####:  658:                        p->tbl_p++;
    #####:  659:                        continue;
        -:  660:                };
        -:  661:
        -:  662:                // load second opcode if needed
        1:  663:                if (IS_SET (p->new_flags, F_OPC2))
        -:  664:                {
        -:  665:                        uint8_t opc2;
    #####:  666:                        if (Da_stage1_get_next_byte(p, &opc2)==false)
        -:  667:                        {
    #####:  668:                                if (dbg_print)
    #####:  669:                                        printf ("%s() line %d\n",
        __func__, __LINE__);
    #####:  670:                                return false;
        -:  671:                        };
...
```

This is how many times each line was executed during run. ##### means never. 1 means one.

Now the goal: to execute all lines at least once, with the help of as few tests as possible. This is how the problem can be stated in plain English language:

for the line X, the test Y OR the test Z or the test M must be ran.

We generate such (hard) clauses for each lines.

Also, we say to MaxSAT solver to find such a solution, where as few tests would be True, as possible. And this is my Python program to do so:

```
#!/usr/bin/env python3

import re, sys, os

# read gcov result and return a list of lines executed:
def parse_gcov_file (fname):
    lines=[]
    f=open(fname,"r")

    while True:
        l=f.readline().rstrip()
        m = re.search('^ *([0-9]+):  ([0-9]+):.*$', l)
        if m!=None:
            lines.append(int(m.group(2)))
        if len(l)==0:
            break

    f.close()
    return lines

max_test_n=0

# k=line, v=list of tests
lines={}

# enumerate all gcov-files:
for (dirpath, dirnames, filenames) in os.walk ("gcovs"):
    for fname in filenames:
        #print "c fname", fname
        fullfname="gcovs/"+fname
        test_n=int(fname.split(".")[0])
        max_test_n=max(max_test_n, test_n)

        lines_executed=parse_gcov_file (fullfname)

        for line in lines_executed:
            lines[line]=lines.get(line, [])+[test_n]

def list_to_CNF(l):
    return "10000 "+" ".join(map(str, l))+" 0"
```

```
print ("p wcnf "+str(max_test_n)+" "+str(len(lines)+max_test_n-1)+" 10000")

# hard clauses. each MUST be satisfied
# "test_1 OR test_2 OR ..." for each line
print ("c lines:")
for line in lines:
    print ("c line=", str(line))
    print (list_to_CNF(lines[line]))

# soft clauses. as many should be satisfied (be False) as possible
for test_n in range(max_test_n):
    print ("1 -"+str(test_n+1)+" 0")
```

I run it:

```
python minimize_tests1.py > 1.wcnf
```

The resulting WCNF[5] file is like:

```
p wcnf 12532 12832 10000
c lines:
c line= 768
10000 3432 6250 9716 9034 2020 4138 1308 5546 3433 4137 2728 7661 12449 9033 10400
    4842 9717 4841 5547 8351 11083 2729 6955 11766 12450 2021 1309 6251 11767 6956
    10399 11084 8350 7662 0
c line= 375
10000 1126 0
c line= 521
10000 12411 1796 8139 11130 2620 7728 11380 6631 1774 7685 5381 10996 626 6252 8189
    4546 6191 11398 3865 ...
...
1 -1 0
1 -2 0
1 -3 0
1 -4 0
...
1 -12528 0
1 -12529 0
1 -12530 0
1 -12531 0
1 -12532 0
```

”c” is comment. I’m putting line (of code) number here. So for line 768, any tests among 3432 6250 9716 9034 2020 4138 1308 5546 3433 4137 2728 7661 must be present, at least one. For line 375, a (single) test 1126 must be present.

(By the way, it’s quite interesting to see what line of code is triggered by a *single* test. Surely, they are must be present in minimized test suite.)

All test numbers are then enumerated as soft constraints.

Now I’m running the Open-WBO MaxSAT solver on the given WCNF file:

```
open-wbo 1.wcnf > result
```

```
c
c Open-WBO:      a Modular MaxSAT Solver -- based on Glucose4.1 (core version)
...

s OPTIMUM FOUND
v -1 -2 -3 -4 -5 -6 -7 -8 -9 -10 -11 -12 -13 -14 -15 -16 -17 -18 -19 -20 -21 ...
    -1125 1126 -1127 ...
...
```

[5] Weighted Conjunctive normal form

The output consists of all variables. But since a test number is mapped to a line number, this number also mapped to a variable's number. Efficiently, Open-WBO reports, which tests are to be picked (like 1126th).

I wrote an utility for that:

```
#!/usr/bin/env python3
import sys

with open("result") as f:
    content = f.readlines()
content = [x.strip() for x in content]

with open("tests.h") as f:
    tests = f.readlines()
tests = [x.strip() for x in tests]

for test in content[-1][2:].split(" "):
    if test.startswith("-"):
        continue
    print (tests[int(test)-1])
```

And we found that to cover (almost) all lines of code in my disassembler, only 11 tests are enough!

```
if (test_all || line==__LINE__) disas_test1(Fuzzy_False, (const byte*)"\x64\xA3\x12\
    x34\x56\x78", 0x100, "FS: MOV [78563412h], EAX");
if (test_all || line==__LINE__) disas_test1(Fuzzy_False, (const byte*)"\xD9\xEE", 0
    x123456, "FLDZ");
if (test_all || line==__LINE__) disas_test1(Fuzzy_False, (const byte*)"\xF0\x66\x0F\
    xB1\x0B", 0x1234, "LOCK CMPXCHG [EBX], CX");
if (test_all || line==__LINE__) disas_test1(Fuzzy_True, (const byte*)"\x44\xAB", 0
    x4f5b, "STOSD");
if (test_all || line==__LINE__) disas_test1(Fuzzy_True, (const byte*)"\x44\xBC\x90\
    x90\x90\x90", 0x507c, "MOV ESP, 90909090h");
if (test_all || line==__LINE__) disas_test1(Fuzzy_True, (const byte*)"\x45\xDB\x90\
    x90\x90\x90\x90", 0x638b, "FIST [R8-6f6f6f70h]");
if (test_all || line==__LINE__) disas_test1(Fuzzy_True, (const byte*)"\x4B\xC8\x90\
    x90\x90", 0xc848, "ENTER 0FFFFFFFFFFFF9090h, 90h");
if (test_all || line==__LINE__) disas_test1(Fuzzy_True, (const byte*)"\x4F\xBF\x90\
    x90\x90\x90\x90\x90\x90\x90", 0x10baf, "MOV R15, 9090909090909090h");
if (test_all || line==__LINE__) disas_test1(Fuzzy_True, (const byte*)"\x66\x0F\x38\
    x3D\xD0", 3, "PMAXSD XMM2, XMM0");
if (test_all || line==__LINE__) disas_test1(Fuzzy_True, (const byte*)"\xF2\x0F\x5A\
    xCE", 0, "CVTSD2SS XMM1, XMM6");
if (test_all || line==__LINE__) disas_test1(Fuzzy_True, (const byte*)"\xF3\x42\x0F\
    x11\x74\x1D\x98", 3, "MOVSS [RBP+R11-68h], XMM6");
```

(It's important to know that my "bloated" tests was not perfect, some lines of code like error reporting are like "dead code" now, but it's OK.)

At this point, my readers perhaps could stop reading and reuse my ideas for their own code and tests.

But... As I mentioned, a disassembler has a *mega-table*. And we want to touch each its entry during tests at least once, like each line of code. And this is a second version of my program:

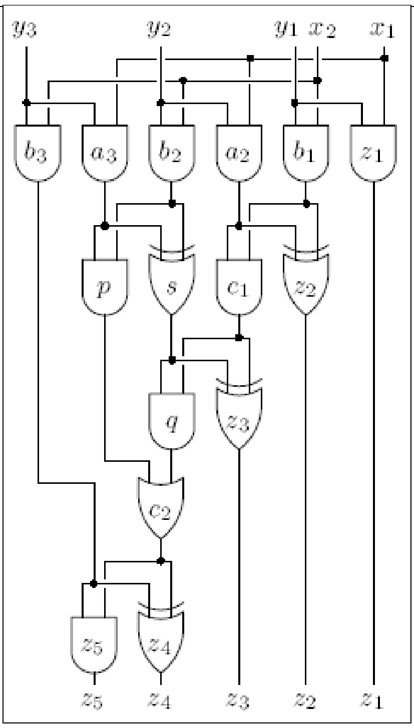
```
#!/usr/bin/env python3

import re, sys, os

# read gcov result and return a list of lines executed:
def parse_gcov_file (fname):
    lines=[]
    f=open(fname,"r")

    while True:
        l=f.readline().rstrip()
        m = re.search('^ *([0-9]+):  ([0-9]+):.*$', l)
        if m!=None:
            lines.append(int(m.group(2)))
        if len(l)==0:
            break

    f.close()
    return lines

max_test_n=0

# k=line, v=list of tests
lines={}

# k=tbl_entry, v=list of tests
tbl_entries={}

# enumerate all gcov-files:
for (dirpath, dirnames, filenames) in os.walk ("gcovs"):
    for fname in filenames:
        #print "c fname", fname
        fullfname="gcovs/"+fname
        test_n=int(fname.split(".")[0])
        max_test_n=max(max_test_n, test_n)
        tbl_entry=int(fname.split(".")[1])

        tbl_entries[tbl_entry]=tbl_entries.get(tbl_entry, [])+[test_n]

        lines_executed=parse_gcov_file (fullfname)

        for line in lines_executed:
            lines[line]=lines.get(line, [])+[test_n]

def list_to_CNF(l):
    return "10000 "+" ".join(map(str, l))+" 0"

print ("p wcnf "+str(max_test_n)+" "+str(len(tbl_entries)+len(lines)+max_test_n-1)+"
   10000")

# hard clauses. each MUST be satisfied
# "test_1 OR test_2 OR ..." for table entry
print ("c tbl entries:")
for tbl_entry in tbl_entries:
    print ("c tbl_entry="+str(tbl_entry))
    print (list_to_CNF(tbl_entries[tbl_entry]))

print ("c lines:")
for line in lines:
    print ("c line=", str(line))
    print (list_to_CNF(lines[line]))

# soft clauses. as many should be satisfied (be False) as possible
for test_n in range(max_test_n):
    print ("1 -"+str(test_n+1)+" 0")
```

Now only 301 tests are enough to cover (almost) all lines in my disassembler and to touch (almost) all entries in the *mega-table*. Much better than 12531.

Also, Open-WBO seems to be a better tool for the job, it works faster than Z3. Or maybe Z3 can be tuned?

14.3 Fault check of digital circuit

Donald Knuth's TAOCP section 7.2.2.2 has the following exercise.



Find a way to check, if it was soldered correctly, with no wires stuck at ground (always 0) or current (always 1). You can just enumerate all possible inputs (5) and this will be a table of correct inputs/outputs, 32 pairs. But you want to make fault check as fast as possible and minimize test set.

This is almost a problem I've been writing before: 14.1.

We want such a test set, so that all gates' outputs will output 0 and 1, at least once. And the test set should be as small, as possible.

The source code is very close to my previous example...

```
#!/usr/bin/env python3
from z3 import *

# 5 inputs, so 1<<5=32 posible combinations:
TOTAL_TESTS=1<<5

# number of gates and/or gates' outputs:
OUTPUTS_TOTAL=15

OUT_Z1, OUT_Z2, OUT_Z3, OUT_Z4, OUT_Z5, OUT_A2, OUT_A3, OUT_B1, OUT_B2, OUT_B3,
    OUT_C1, OUT_C2, OUT_P, OUT_Q, OUT_S = range(OUTPUTS_TOTAL)

out_false_if={}
out_true_if={}

# enumerate all possible inputs
for i in range(1<<5):
    x1=i&1
    x2=(i>>1)&1
    y1=(i>>2)&1
    y2=(i>>3)&1
    y3=(i>>4)&1

    outs={}

    # simulate the circuit:
    outs[OUT_Z1]=y1&x1
    outs[OUT_B1]=y1&x2
    outs[OUT_A2]=x1&y2
    outs[OUT_B2]=y2&x2
    outs[OUT_A3]=x1&y3
    outs[OUT_B3]=y3&x2

    outs[OUT_P]=outs[OUT_A3] & outs[OUT_B2]
    outs[OUT_S]=outs[OUT_A3] ^ outs[OUT_B2]
    outs[OUT_C1]=outs[OUT_A2] & outs[OUT_B1]
    outs[OUT_Z2]=outs[OUT_A2] ^ outs[OUT_B1]

    outs[OUT_Q]=outs[OUT_S] & outs[OUT_C1]
    outs[OUT_Z3]=outs[OUT_S] ^ outs[OUT_C1]

    outs[OUT_C2]=outs[OUT_P] | outs[OUT_Q]

    outs[OUT_Z5]=outs[OUT_B3] & outs[OUT_C2]
    outs[OUT_Z4]=outs[OUT_B3] ^ outs[OUT_C2]

    inputs=(y3, y2, y1, x2, x1)
    print ("inputs:", inputs, "outputs of all gates:", outs)

    for o in range(OUTPUTS_TOTAL):
        if outs[o]==0:
            if o not in out_false_if:
                out_false_if[o]=[]
            out_false_if[o].append(i)
        else:
            if o not in out_true_if:
                out_true_if[o]=[]
            out_true_if[o].append(i)

for o in range(OUTPUTS_TOTAL):
    print ("output #%d" % o)
    print ("    false if:", out_false_if[o])
    print ("    true if:", out_true_if[o])

s=Solver()

# if the test will be picked or not:
tests=[Int('test_%d' % (t)) for t in range(TOTAL_TESTS)]

# this is optimization problem:
opt = Optimize()

# a test may be picked (1) or not (0):
for t in tests:
    opt.add(Or(t==0, t==1))

# this generates expression like (tests[0]==1 OR tests[1]==1 OR tests[X]==1):
for o in range(OUTPUTS_TOTAL):
    opt.add(Or(*[tests[i]==1 for i in out_false_if[o]]))
    opt.add(Or(*[tests[i]==1 for i in out_true_if[o]]))

# minimize number of tests:
opt.minimize(Sum(*tests))

print (opt.check())
m=opt.model()

for i in range(TOTAL_TESTS):
    t=m[tests[i]].as_long()
    if t==1:
        print (format(i, '05b'))
```

The output:

```
inputs: (0, 0, 0, 0, 0) outputs of all gates: {0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6:
    0, 7: 0, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0, 13: 0, 14: 0}
inputs: (0, 0, 0, 0, 1) outputs of all gates: {0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6:
    0, 7: 0, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0, 13: 0, 14: 0}
inputs: (0, 0, 0, 1, 0) outputs of all gates: {0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6:
    0, 7: 0, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0, 13: 0, 14: 0}
inputs: (0, 0, 0, 1, 1) outputs of all gates: {0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6:
    0, 7: 0, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0, 13: 0, 14: 0}
inputs: (0, 0, 1, 0, 0) outputs of all gates: {0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6:
    0, 7: 0, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0, 13: 0, 14: 0}
inputs: (0, 0, 1, 0, 1) outputs of all gates: {0: 1, 1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6:
    0, 7: 0, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0, 13: 0, 14: 0}
inputs: (0, 0, 1, 1, 0) outputs of all gates: {0: 0, 1: 1, 2: 0, 3: 0, 4: 0, 5: 0, 6:
    0, 7: 1, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0, 13: 0, 14: 0}
inputs: (0, 0, 1, 1, 1) outputs of all gates: {0: 1, 1: 1, 2: 0, 3: 0, 4: 0, 5: 0, 6:
    0, 7: 1, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0, 13: 0, 14: 0}
inputs: (0, 1, 0, 0, 0) outputs of all gates: {0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6:
    0, 7: 0, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0, 13: 0, 14: 0}
inputs: (0, 1, 0, 0, 1) outputs of all gates: {0: 0, 1: 1, 2: 0, 3: 0, 4: 0, 5: 1, 6:
    0, 7: 0, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0, 13: 0, 14: 0}
inputs: (0, 1, 0, 1, 0) outputs of all gates: {0: 0, 1: 0, 2: 1, 3: 0, 4: 0, 5: 0, 6:
    0, 7: 0, 8: 1, 9: 0, 10: 0, 11: 0, 12: 0, 13: 0, 14: 1}
inputs: (0, 1, 0, 1, 1) outputs of all gates: {0: 0, 1: 1, 2: 1, 3: 0, 4: 0, 5: 1, 6:
    0, 7: 0, 8: 1, 9: 0, 10: 0, 11: 0, 12: 0, 13: 0, 14: 1}
inputs: (0, 1, 1, 0, 0) outputs of all gates: {0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6:
    0, 7: 0, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0, 13: 0, 14: 0}
inputs: (0, 1, 1, 0, 1) outputs of all gates: {0: 1, 1: 1, 2: 0, 3: 0, 4: 0, 5: 1, 6:
    0, 7: 0, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0, 13: 0, 14: 0}
inputs: (0, 1, 1, 1, 0) outputs of all gates: {0: 0, 1: 1, 2: 1, 3: 0, 4: 0, 5: 0, 6:
    0, 7: 1, 8: 1, 9: 0, 10: 0, 11: 0, 12: 0, 13: 0, 14: 1}
inputs: (0, 1, 1, 1, 1) outputs of all gates: {0: 1, 1: 0, 2: 0, 3: 1, 4: 0, 5: 1, 6:
    0, 7: 1, 8: 1, 9: 0, 10: 1, 11: 1, 12: 0, 13: 1, 14: 1}
inputs: (1, 0, 0, 0, 0) outputs of all gates: {0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6:
    0, 7: 0, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0, 13: 0, 14: 0}
inputs: (1, 0, 0, 0, 1) outputs of all gates: {0: 0, 1: 0, 2: 1, 3: 0, 4: 0, 5: 0, 6:
    1, 7: 0, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0, 13: 0, 14: 1}
inputs: (1, 0, 0, 1, 0) outputs of all gates: {0: 0, 1: 0, 2: 0, 3: 1, 4: 0, 5: 0, 6:
    0, 7: 0, 8: 0, 9: 1, 10: 0, 11: 0, 12: 0, 13: 0, 14: 0}
inputs: (1, 0, 0, 1, 1) outputs of all gates: {0: 0, 1: 0, 2: 1, 3: 1, 4: 0, 5: 0, 6:
    1, 7: 0, 8: 0, 9: 1, 10: 0, 11: 0, 12: 0, 13: 0, 14: 1}
inputs: (1, 0, 1, 0, 0) outputs of all gates: {0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6:
    0, 7: 0, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0, 13: 0, 14: 0}
inputs: (1, 0, 1, 0, 1) outputs of all gates: {0: 1, 1: 0, 2: 1, 3: 0, 4: 0, 5: 0, 6:
    1, 7: 0, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0, 13: 0, 14: 1}
inputs: (1, 0, 1, 1, 0) outputs of all gates: {0: 0, 1: 1, 2: 0, 3: 1, 4: 0, 5: 0, 6:
    0, 7: 1, 8: 0, 9: 1, 10: 0, 11: 0, 12: 0, 13: 0, 14: 0}
inputs: (1, 0, 1, 1, 1) outputs of all gates: {0: 1, 1: 1, 2: 1, 3: 1, 4: 0, 5: 0, 6:
    1, 7: 1, 8: 0, 9: 1, 10: 0, 11: 0, 12: 0, 13: 0, 14: 1}
inputs: (1, 1, 0, 0, 0) outputs of all gates: {0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6:
    0, 7: 0, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0, 13: 0, 14: 0}
inputs: (1, 1, 0, 0, 1) outputs of all gates: {0: 0, 1: 1, 2: 1, 3: 0, 4: 0, 5: 1, 6:
    1, 7: 0, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0, 13: 0, 14: 1}
inputs: (1, 1, 0, 1, 0) outputs of all gates: {0: 0, 1: 0, 2: 1, 3: 1, 4: 0, 5: 0, 6:
    0, 7: 0, 8: 1, 9: 1, 10: 0, 11: 0, 12: 0, 13: 0, 14: 1}
inputs: (1, 1, 0, 1, 1) outputs of all gates: {0: 0, 1: 1, 2: 0, 3: 0, 4: 1, 5: 1, 6:
    1, 7: 0, 8: 1, 9: 1, 10: 0, 11: 1, 12: 1, 13: 0, 14: 0}
inputs: (1, 1, 1, 0, 0) outputs of all gates: {0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6:
    0, 7: 0, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0, 13: 0, 14: 0}
inputs: (1, 1, 1, 0, 1) outputs of all gates: {0: 1, 1: 1, 2: 1, 3: 0, 4: 0, 5: 1, 6:
    1, 7: 0, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0, 13: 0, 14: 1}
inputs: (1, 1, 1, 1, 0) outputs of all gates: {0: 0, 1: 1, 2: 1, 3: 1, 4: 0, 5: 0, 6:
    0, 7: 1, 8: 1, 9: 1, 10: 0, 11: 0, 12: 0, 13: 0, 14: 1}
inputs: (1, 1, 1, 1, 1) outputs of all gates: {0: 1, 1: 0, 2: 1, 3: 0, 4: 1, 5: 1, 6:
    1, 7: 1, 8: 1, 9: 1, 10: 1, 11: 1, 12: 1, 13: 0, 14: 0}
output #0
    false if: [0, 1, 2, 3, 4, 6, 8, 9, 10, 11, 12, 14, 16, 17, 18, 19, 20, 22, 24,
       25, 26, 27, 28, 30]
    true if: [5, 7, 13, 15, 21, 23, 29, 31]
output #1
    false if: [0, 1, 2, 3, 4, 5, 8, 10, 12, 15, 16, 17, 18, 19, 20, 21, 24, 26, 28,
       31]
    true if: [6, 7, 9, 11, 13, 14, 22, 23, 25, 27, 29, 30]
output #2
    false if: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 12, 13, 15, 16, 18, 20, 22, 24, 27, 28]
    true if: [10, 11, 14, 17, 19, 21, 23, 25, 26, 29, 30, 31]
output #3
    false if: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16, 17, 20, 21, 24,
       25, 27, 28, 29, 31]
    true if: [15, 18, 19, 22, 23, 26, 30]
output #4
    false if: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
       20, 21, 22, 23, 24, 25, 26, 28, 29, 30]
    true if: [27, 31]
output #5
    false if: [0, 1, 2, 3, 4, 5, 6, 7, 8, 10, 12, 14, 16, 17, 18, 19, 20, 21, 22, 23,
        24, 26, 28, 30]
    true if: [9, 11, 13, 15, 25, 27, 29, 31]
output #6
    false if: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 18, 20, 22,
       24, 26, 28, 30]
    true if: [17, 19, 21, 23, 25, 27, 29, 31]
output #7
    false if: [0, 1, 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 16, 17, 18, 19, 20, 21, 24,
       25, 26, 27, 28, 29]
    true if: [6, 7, 14, 15, 22, 23, 30, 31]
output #8
    false if: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 12, 13, 16, 17, 18, 19, 20, 21, 22, 23,
       24, 25, 28, 29]
    true if: [10, 11, 14, 15, 26, 27, 30, 31]
output #9
    false if: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 20, 21,
       24, 25, 28, 29]
    true if: [18, 19, 22, 23, 26, 27, 30, 31]
output #10
    false if: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16, 17, 18, 19, 20,
       21, 22, 23, 24, 25, 26, 27, 28, 29, 30]
    true if: [15, 31]
output #11
    false if: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16, 17, 18, 19, 20,
       21, 22, 23, 24, 25, 26, 28, 29, 30]
    true if: [15, 27, 31]
output #12
    false if: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
       20, 21, 22, 23, 24, 25, 26, 28, 29, 30]
    true if: [27, 31]
output #13
    false if: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16, 17, 18, 19, 20,
       21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31]
    true if: [15]
output #14
```

```
    false if: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 12, 13, 16, 18, 20, 22, 24, 27, 28, 31]
    true if: [10, 11, 14, 15, 17, 19, 21, 23, 25, 26, 29, 30]
sat
01111
10001
11011
```

This is it, you can test this circuit using just 3 test vectors: 01111, 10001 and 11011.

However, Donald Knuth's test set is bigger: 5 test vectors, but his algorithm also checks ”fanout” gates (one input, multiple outputs), which also may be faulty. I've omitted this for simplification.

14.4 GCD and LCM

Untitled collection of math notes[6]has short explanation of GCD and LCM.

14.4.1 GCD

To compute GCD, one of the oldest algorithms is used: Euclidean algorithm. But, I can demonstrate how to make things much less efficient, but more spectacular.

To find GCD of 14 and 8, we are going to solve this system of equations:

```
x*GCD=14
y*GCD=8
```

Then we drop x and y, we don't need them. This system can be solved using a piece of paper and pencil, but GCD must be as big as possible. Here we can use Z3 in MaxSMT mode:

```
#!/usr/bin/env python

from z3 import *

opt = Optimize()

x,y,GCD=Ints('x y GCD')

opt.add(x*GCD==14)
opt.add(y*GCD==8)

h=opt.maximize(GCD)

print (opt.check())
print (opt.model())
```

That works:

```
sat
[y = 4, x = 7, GCD = 2]
```

What if we need to find GCD for 3 numbers? Maybe we are going to fill a space with biggest possible cubes?

```
#!/usr/bin/env python

from z3 import *

opt = Optimize()

x,y,z,GCD=Ints('x y z GCD')

opt.add(x*GCD==300)
opt.add(y*GCD==333)
opt.add(z*GCD==900)

```

[6] https://xn--mxacd.xn--qxam/math-notes.pdf

```
h=opt.maximize(GCD)

print (opt.check())
print (opt.model())
```

This is 3:

```
sat
[z = 300, y = 111, x = 100, GCD = 3]
```

In SMT-LIB form:

```
; checked with Z3 and MK85

; must be 21
; see also: https://www.wolframalpha.com/input/?i=GCD[861,3969,840]

(declare-fun x () (_ BitVec 16))
(declare-fun y () (_ BitVec 16))
(declare-fun z () (_ BitVec 16))
(declare-fun GCD () (_ BitVec 16))

(assert (= (bvmul ((_ zero_extend 16) x) ((_ zero_extend 16) GCD)) (_ bv861 32)))
(assert (= (bvmul ((_ zero_extend 16) y) ((_ zero_extend 16) GCD)) (_ bv3969 32)))
(assert (= (bvmul ((_ zero_extend 16) z) ((_ zero_extend 16) GCD)) (_ bv840 32)))

(maximize GCD)

(check-sat)
(get-model)

; correct result:
;(model
;         (define-fun x () (_ BitVec 16) (_ bv41 16)) ; 0x29
;         (define-fun y () (_ BitVec 16) (_ bv189 16)) ; 0xbd
;         (define-fun z () (_ BitVec 16) (_ bv40 16)) ; 0x28
;         (define-fun GCD () (_ BitVec 16) (_ bv21 16)) ; 0x15
;)
```

14.4.2 Least Common Multiple

To find LCM of 4 and 6, we are going to solve the following diophantine (i.e., allowing only integer solutions) system of equations:

$4x = 6y = LCM$

... where LCM>0 and as small, as possible.

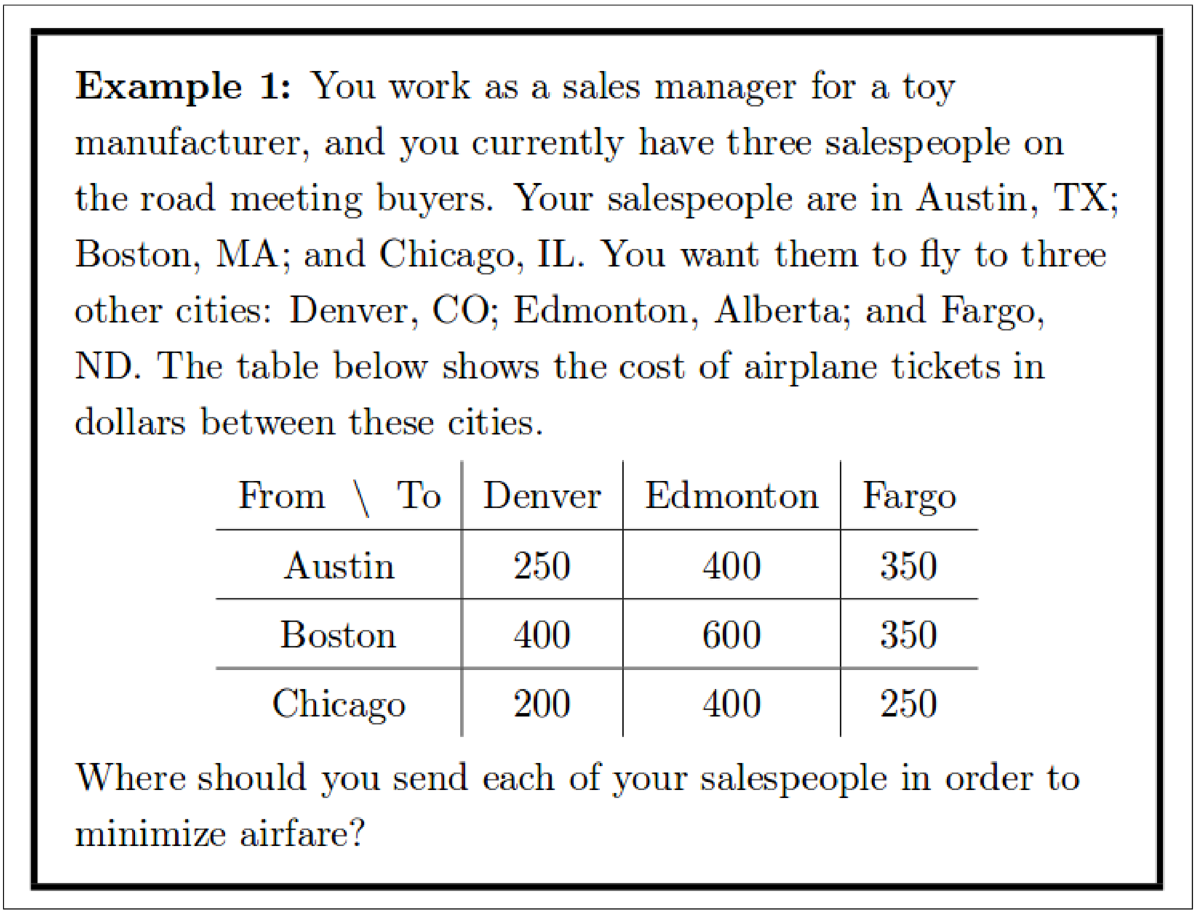
```
#!/usr/bin/env python

from z3 import *

opt = Optimize()

x,y,LCM=Ints('x y LCM')

opt.add(x*4==LCM)
opt.add(y*6==LCM)
opt.add(LCM>0)

h=opt.minimize(LCM)

print (opt.check())
print (opt.model())
```

The (correct) answer:

```
sat
[y = 2, x = 3, LCM = 12]
```

14.5 Assignment problem

I've found this at http://www.math.harvard.edu/archive/20_spring_05/handouts/assignment_overheads.pdf and took screenshot:



Example 1: You work as a sales manager for a toy manufacturer, and you currently have three salespeople on the road meeting buyers. Your salespeople are in Austin, TX; Boston, MA; and Chicago, IL. You want them to fly to three other cities: Denver, CO; Edmonton, Alberta; and Fargo, ND. The table below shows the cost of airplane tickets in dollars between these cities.

From \ To	Denver	Edmonton	Fargo
Austin	250	400	350
Boston	400	600	350
Chicago	200	400	250

Where should you send each of your salespeople in order to minimize airfare?

As in my previous examples, Z3 and SMT-solver may be overkill for the task. Simpler algorithm exists for this task (*Hungarian algorithm/method*) [7].

But again, I use it to demonstrate the problem + as SMT-solvers demonstration.

Here is what I do:

```
#!/usr/bin/env python3

from z3 import *

# this is optimization problem:
s=Optimize()

choice=[Int('choice_%d' % i) for i in range(3)]
row_value=[Int('row_value_%d' % i) for i in range(3)]

for i in range(3):
    s.add(And(choice[i]>=0, choice[i]<=2))

s.add(Distinct(choice))
s.add(row_value[0]==
```

[7] See also: https://en.wikipedia.org/wiki/Hungarian_algorithm.

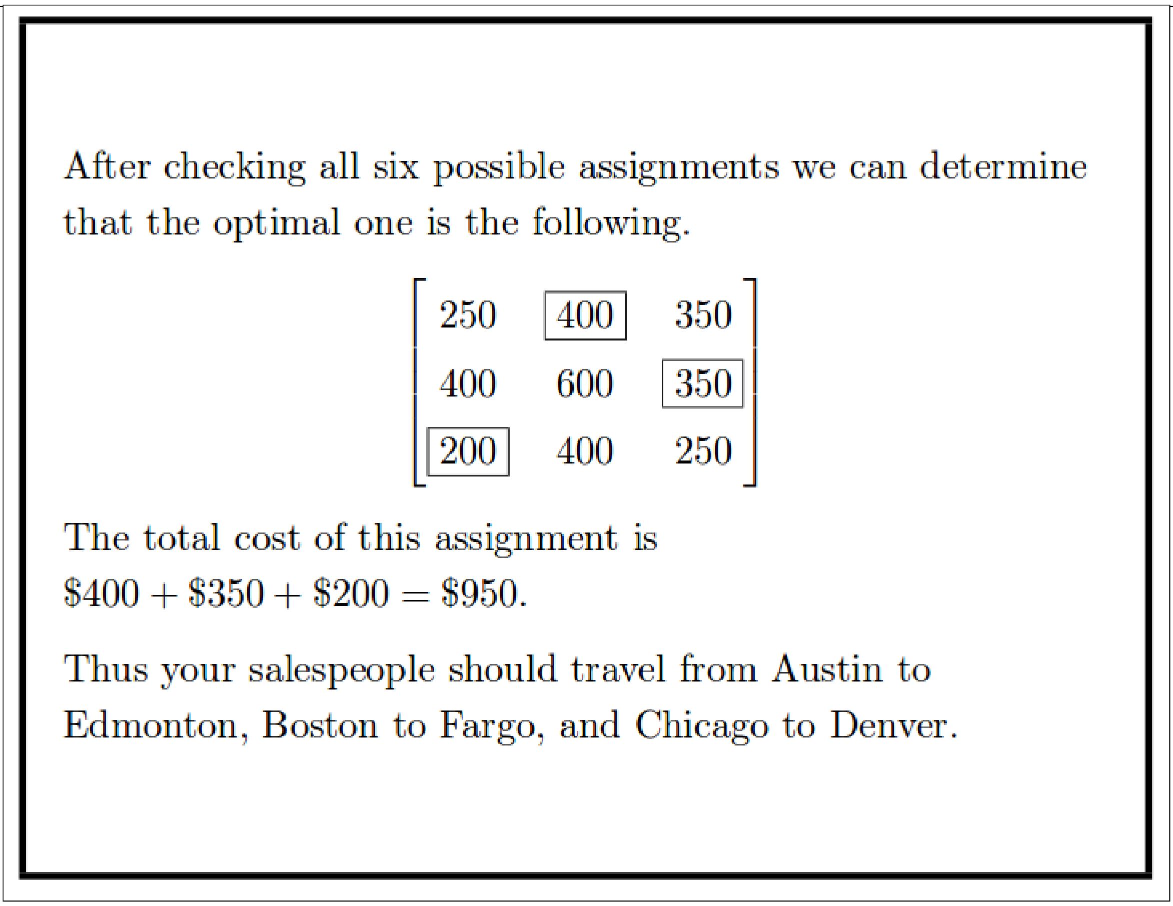
```
        If(choice[0]==0, 250,
        If(choice[0]==1, 400,
        If(choice[0]==2, 350, -1))))

s.add(row_value[1]==
        If(choice[1]==0, 400,
        If(choice[1]==1, 600,
        If(choice[1]==2, 350, -1))))

s.add(row_value[2]==
        If(choice[2]==0, 200,
        If(choice[2]==1, 400,
        If(choice[2]==2, 250, -1))))

final_sum=Int('final_sum')

# final_sum equals to sum of all row_value[] values:
s.add(final_sum==Sum(*row_value))

# find such a (distinct) choice[]'s, so that the final_sum would be minimal:
s.minimize(final_sum)

print (s.check())
print (s.model())
```

In plain English this means *choose such columns, so that their sum would be as small as possible.*

Result is seems to be correct:

```
sat
[choice_0 = 1,
 choice_1 = 2,
 choice_2 = 0,
 z3name!12 = 0,
 z3name!7 = 1,
 z3name!10 = 2,
 z3name!8 = 0,
 z3name!11 = 0,
 z3name!9 = 0,
 final_sum = 950,
 row_value_2 = 200,
 row_value_1 = 350,
 row_value_0 = 400]
```

Again, as it is in the corresponding PDF presentation:



After checking all six possible assignments we can determine that the optimal one is the following.

$$\begin{bmatrix} 250 & \boxed{400} & 350 \\ 400 & 600 & \boxed{350} \\ \boxed{200} & 400 & 250 \end{bmatrix}$$

The total cost of this assignment is $\$400 + \$350 + \$200 = \$950$.

Thus your salespeople should travel from Austin to Edmonton, Boston to Fargo, and Chicago to Denver.

(However, I've no idea what "z3name" variables mean, perhaps, some internal variables?)

The problem can also be stated in SMT-LIB 2.0 format, and solved using MK85:

```
; checked with Z3 and MK85

(declare-fun choice1 () (_ BitVec 2))
(declare-fun choice2 () (_ BitVec 2))
(declare-fun choice3 () (_ BitVec 2))

(declare-fun row_value1 () (_ BitVec 16))
(declare-fun row_value2 () (_ BitVec 16))
(declare-fun row_value3 () (_ BitVec 16))

(declare-fun final_sum () (_ BitVec 16))

(assert (bvule choice1 (_ bv3 2)))
(assert (bvule choice2 (_ bv3 2)))
(assert (bvule choice3 (_ bv3 2)))

(assert (distinct choice1 choice2 choice3))

(assert (= row_value1
        (ite (= choice1 (_ bv0 2)) (_ bv250 16)
        (ite (= choice1 (_ bv1 2)) (_ bv400 16)
        (ite (= choice1 (_ bv2 2)) (_ bv250 16)
        (_ bv999 16))))))

(assert (= row_value2
        (ite (= choice2 (_ bv0 2)) (_ bv400 16)
        (ite (= choice2 (_ bv1 2)) (_ bv600 16)
        (ite (= choice2 (_ bv2 2)) (_ bv350 16)
        (_ bv999 16))))))

(assert (= row_value3
        (ite (= choice3 (_ bv0 2)) (_ bv200 16)
        (ite (= choice3 (_ bv1 2)) (_ bv400 16)
        (ite (= choice3 (_ bv2 2)) (_ bv250 16)
        (_ bv999 16))))))

(assert (= final_sum (bvadd row_value1 row_value2 row_value3)))

(minimize final_sum)

(check-sat)
(get-model)
```

Listing 14.1: The result

```
sat
(model
        (define-fun choice1 () (_ BitVec 2) (_ bv1 2)) ; 0x1
        (define-fun choice2 () (_ BitVec 2) (_ bv2 2)) ; 0x2
        (define-fun choice3 () (_ BitVec 2) (_ bv0 2)) ; 0x0
        (define-fun row_value1 () (_ BitVec 16) (_ bv400 16)) ; 0x190
        (define-fun row_value2 () (_ BitVec 16) (_ bv350 16)) ; 0x15e
        (define-fun row_value3 () (_ BitVec 16) (_ bv200 16)) ; 0xc8
        (define-fun final_sum () (_ BitVec 16) (_ bv950 16)) ; 0x3b6
)
```

14.6 Find maximal clique using MaxSAT solver

Untitled collection of math notes[8]has a short intro to graph cliques.

Though not the most efficient method, but very spectacular and instructive.

In short: if there is no edge between a pair of vertices, these two vertices cannot be included in the solution simultaneously.

Given the 50-vertices graph: https://smt.st/current_tree/MaxSxT/clique_openwbo/edges.txt.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import my_utils, SAT_lib
import sys, re

VERTICES=50

s=SAT_lib.SAT_lib(maxsat=True)

f=open(sys.argv[1],"r")

vertices=[s.create_var() for v in range(VERTICES)]
# as much verties, as possible:
for v in range(VERTICES):
    s.fix_soft_always_true(vertices[v], 1)

edges=set()
while True:
    raw=f.readline()
    l=raw.rstrip ()
    if len(l)==0:
        break
```

[8] https://xn--mxacd.xn--qxam/math-notes.pdf

```
    m=re.search('UndirectedEdge\[(.*), (.*)\]', l)
    if m!=None:
        v1=int(m.group(1))-1
        v2=int(m.group(2))-1
        edges.add((v1,v2))
        edges.add((v2,v1))

for i in range(VERTICES):
    for j in range(VERTICES):
        if i==j:
            continue
        if (i,j) not in edges:
            # if there is no edge between a pair of vertices,
            # these two vertices cannot be included in the solution simultaneously:
            s.add_clause([s.neg(vertices[i]), s.neg(vertices[j])])

f.close()

print ("first 10 solutions:")
cnt=0
while s.solve():
    solution=[]
    for v in range(VERTICES):
        val=s.get_var_from_solution(vertices[v])
        if val!=0:
            solution.append(v+1)
    print (f"clique size {len(solution)}: {solution}")
    cnt+=1
    if cnt==10:
        break
    s.add_negated_solution_as_clause()
```

Run:

```
python3 1.py edges.txt
```

The resulting WCNF file:

```
p wcnf 52 1502 10000

...

c soft clauses: vertices:
1 3 0
1 4 0
1 5 0
1 6 0
...
1 49 0
1 50 0
1 51 0
1 52 0

c hard clauses: list of all non-existent pairs:
10000 -3 -5 0
10000 -3 -6 0
10000 -3 -11 0
10000 -3 -13 0
10000 -3 -14 0
10000 -3 -15 0
10000 -3 -17 0
10000 -3 -19 0
```

```
...
10000 -52 -31 0
10000 -52 -34 0
10000 -52 -36 0
10000 -52 -39 0
10000 -52 -40 0
10000 -52 -45 0
10000 -52 -46 0
10000 -52 -47 0
10000 -52 -48 0
```

And the (correct) result, however, not the single one:

```
first 10 solutions:
clique size 6: [6, 7, 13, 22, 36, 42]
clique size 6: [20, 25, 33, 37, 39, 49]
clique size 6: [20, 30, 33, 39, 49, 50]
clique size 6: [20, 25, 33, 39, 49, 50]
clique size 6: [6, 7, 13, 33, 42, 46]
clique size 6: [6, 13, 24, 33, 46, 48]
clique size 6: [6, 13, 24, 33, 42, 46]
clique size 6: [4, 11, 22, 47, 48, 50]
clique size 6: [5, 8, 11, 19, 24, 31]
clique size 6: [1, 2, 7, 16, 36, 49]
```

Further reading: https://cs.stackexchange.com/questions/70531/reduction-3sat-and-clique.

14.7 ”Polite customer” problem

This is probably not relevant in the era of credit cards, but...

In a shop or bar, you have to pay something in cash and, as a polite customer, you try to give such a set of banknotes, so that the cashier will not touch small banknotes in his/her cash desk, which are valuable (because of impoliteness of many others).

For a small amount of money, this can be solved in one's mind, but for large...

The problem is surprisingly easy to solve using Z3 SMT-solver:

```
#!/usr/bin/env python3
from z3 import *

# we don't use coins for simplicity, but it's not a problem to add them

# banknotes in cash register
cash_register_1=1
cash_register_3=0
cash_register_5=2
cash_register_10=2
cash_register_25=0
cash_register_50=12
cash_register_100=10

# ... in customer's wallet
customer_wallet_1=2
customer_wallet_3=1
customer_wallet_5=1
customer_wallet_10=1
customer_wallet_25=0
customer_wallet_50=15
customer_wallet_100=20

# what customer have to pay
checkout=2135
```

```
from_cash_register_1=Int('from_cash_register_1')
from_cash_register_3=Int('from_cash_register_3')
from_cash_register_5=Int('from_cash_register_5')
from_cash_register_10=Int('from_cash_register_10')
from_cash_register_25=Int('from_cash_register_25')
from_cash_register_50=Int('from_cash_register_50')
from_cash_register_100=Int('from_cash_register_100')

from_customer_wallet_1=Int('from_customer_wallet_1')
from_customer_wallet_3=Int('from_customer_wallet_3')
from_customer_wallet_5=Int('from_customer_wallet_5')
from_customer_wallet_10=Int('from_customer_wallet_10')
from_customer_wallet_25=Int('from_customer_wallet_25')
from_customer_wallet_50=Int('from_customer_wallet_50')
from_customer_wallet_100=Int('from_customer_wallet_100')

s=Optimize()

# banknotes pulled from cash_register are limited by 0 and what is defined:
s.add(And(from_cash_register_1 >= 0, from_cash_register_1 <= cash_register_1))
s.add(And(from_cash_register_3 >= 0, from_cash_register_3 <= cash_register_3))
s.add(And(from_cash_register_5 >= 0, from_cash_register_5 <= cash_register_5))
s.add(And(from_cash_register_10 >= 0, from_cash_register_10 <= cash_register_10))
s.add(And(from_cash_register_25 >= 0, from_cash_register_25 <= cash_register_25))
s.add(And(from_cash_register_50 >= 0, from_cash_register_50 <= cash_register_50))
s.add(And(from_cash_register_100 >= 0, from_cash_register_100 <= cash_register_100))

# same for customer's wallet:
s.add(And(from_customer_wallet_1 >= 0, from_customer_wallet_1 <= customer_wallet_1))
s.add(And(from_customer_wallet_3 >= 0, from_customer_wallet_3 <= customer_wallet_3))
s.add(And(from_customer_wallet_5 >= 0, from_customer_wallet_5 <= customer_wallet_5))
s.add(And(from_customer_wallet_10 >= 0, from_customer_wallet_10 <= customer_wallet_10
    ))
s.add(And(from_customer_wallet_25 >= 0, from_customer_wallet_25 <= customer_wallet_25
    ))
s.add(And(from_customer_wallet_50 >= 0, from_customer_wallet_50 <= customer_wallet_50
    ))
s.add(And(from_customer_wallet_100 >= 0, from_customer_wallet_100 <=
    customer_wallet_100))

from_cash_register=Int('from_cash_register')

# sum:
s.add(from_cash_register==
from_cash_register_1*1 +
from_cash_register_3*3 +
from_cash_register_5*5 +
from_cash_register_10*10 +
from_cash_register_25*25 +
from_cash_register_50*50 +
from_cash_register_100*100)

from_customer=Int('from_customer')

s.add(from_customer==
from_customer_wallet_1*1 +
from_customer_wallet_3*3 +
from_customer_wallet_5*5 +
from_customer_wallet_10*10 +
from_customer_wallet_25*25 +
from_customer_wallet_50*50 +
from_customer_wallet_100*100)

# the main constraint:
s.add(from_customer - checkout == from_cash_register)

value_of_banknotes_from_cash_register=Int('value_of_banknotes_from_cash_register')

# cashiers value small banknotes more:
s.add(value_of_banknotes_from_cash_register==
from_cash_register_1*100 +
from_cash_register_3*50 +
from_cash_register_5*20 +
from_cash_register_10*10 +
from_cash_register_25*5 +
from_cash_register_50*2 +
from_cash_register_100)

# try to minimize this value during transaction:
s.minimize(value_of_banknotes_from_cash_register)

print (s.check())
m=s.model()

print ("from_cash_register_1=", m[from_cash_register_1])
print ("from_cash_register_3=", m[from_cash_register_3])
print ("from_cash_register_5=", m[from_cash_register_5])
print ("from_cash_register_10=", m[from_cash_register_10])
print ("from_cash_register_25=", m[from_cash_register_25])
print ("from_cash_register_50=", m[from_cash_register_50])
print ("from_cash_register_100=", m[from_cash_register_100])

print ("from_cash_register=", m[from_cash_register])

print ("")

print ("from_customer_wallet_1=", m[from_customer_wallet_1])
print ("from_customer_wallet_3=", m[from_customer_wallet_3])
print ("from_customer_wallet_5=", m[from_customer_wallet_5])
print ("from_customer_wallet_10=", m[from_customer_wallet_10])
print ("from_customer_wallet_25=", m[from_customer_wallet_25])
print ("from_customer_wallet_50=", m[from_customer_wallet_50])
print ("from_customer_wallet_100=", m[from_customer_wallet_100])

print ("from_customer=", m[from_customer])
```

```
sat
from_cash_register_1= 0
from_cash_register_3= 0
from_cash_register_5= 0
from_cash_register_10= 2
from_cash_register_25= 0
from_cash_register_50= 0
from_cash_register_100= 0
from_cash_register= 20

from_customer_wallet_1= 2
from_customer_wallet_3= 1
from_customer_wallet_5= 0
from_customer_wallet_10= 0
from_customer_wallet_25= 0
from_customer_wallet_50= 11
```

```
from_customer_wallet_100= 16
from_customer= 2155
```

So that the cashier have to pull only 2 tenners from cash desk.

Now what if we don't care about small banknotes, but want to make transaction as paperless as possible:

```
...

# minimize number of all banknotes in transaction:
s.minimize(
from_cash_register_1 +
from_cash_register_3 +
from_cash_register_5 +
from_cash_register_10 +
from_cash_register_25 +
from_cash_register_50 +
from_cash_register_100 +
from_customer_wallet_1 +
from_customer_wallet_3 +
from_customer_wallet_5 +
from_customer_wallet_10 +
from_customer_wallet_25 +
from_customer_wallet_50 +
from_customer_wallet_100)

...
```

```
sat
from_cash_register_1= 0
from_cash_register_3= 0
from_cash_register_5= 1
from_cash_register_10= 1
from_cash_register_25= 0
from_cash_register_50= 0
from_cash_register_100= 0
from_cash_register= 15

from_customer_wallet_1= 0
from_customer_wallet_3= 0
from_customer_wallet_5= 0
from_customer_wallet_10= 0
from_customer_wallet_25= 0
from_customer_wallet_50= 3
from_customer_wallet_100= 20
from_customer= 2150
```

Likewise, you can minimize banknotes from cashier only, or from customer only.

14.8 Packing students into a dorm

Given, say, 15 students. And they all have various interests/hobbies in their life, like hiking, clubbing, dancing, swimming, maybe hanging out with girls, etc.

A dormitory has 5 rooms. Three students can be accommodated in each room.

The problem to pack them all in such a way, so that all roommates would share as many interests/hobbies with each other, as possible. To make them happy and tight-knit.

This is what I will do using Open-WBO MaxSAT solver this time and my small Python library.

```
#!/usr/bin/env python3

```

```
import SAT_lib

s=SAT_lib.SAT_lib(maxsat=True)

STUDENTS=15
CEILING_LOG2_STUDENTS=4 # 4 bits to hold a number in 0..14 range

POSSIBLE_INTERESTS=8

# interests/hobbies for each student. 8 are possible:
interests=[
0b10000100,
0b01000001,
0b00000010,
0b01000001,
0b00001001,
0b00101000,
0b00000100,
0b00000011,
0b01000000,
0b00000100,
0b10000001,
0b00001000,
0b00000010,
0b01000001,
0b01000010,
0] # dummy variable, to pad the list to 16 elements

# each variable is "grounded" to the bitmask from interests[]:
interests_vars=[s.alloc_BV(POSSIBLE_INTERESTS) for i in range(2**
    CEILING_LOG2_STUDENTS)]
for st in range(2**CEILING_LOG2_STUDENTS):
    s.fix_BV(interests_vars[st], SAT_lib.n_to_BV(interests[st], POSSIBLE_INTERESTS))

# where each student is located after permutation:
students_positions=[s.alloc_BV(CEILING_LOG2_STUDENTS) for i in range(2**
    CEILING_LOG2_STUDENTS)]

# permutation. all positions are distinct, of course:
s.make_distinct_BVs (students_positions)

# connect interests of each student to permuted version
# use multiplexer...
interests_of_student_in_position={}
for st in range(2**CEILING_LOG2_STUDENTS):
    interests_of_student_in_position[st]=s.create_wide_MUX (interests_vars,
        students_positions[st])

# divide result of permutation by triplets
# we want as many similar bits in interests[] between all 3 students, as possible:
for group in range(int(STUDENTS/3)):
    i1=interests_of_student_in_position[group*3]
    i2=interests_of_student_in_position[group*3+1]
    i3=interests_of_student_in_position[group*3+2]
    s.fix_soft_always_true_all_bits_in_BV(s.BV_AND(i1, i2), weight=1)
    s.fix_soft_always_true_all_bits_in_BV(s.BV_AND(i1, i3), weight=1)
    s.fix_soft_always_true_all_bits_in_BV(s.BV_AND(i2, i3), weight=1)

assert s.solve()

def POPCNT(v):
    rt=0
    for i in range(POSSIBLE_INTERESTS):
        if ((v>>i)&1)==1:
            rt=rt+1
    return rt

total_in_all_groups=0

# print solution:
for group in range(int(STUDENTS/3)):
    print ("* group", group)
    st1=SAT_lib.BV_to_number(s.get_BV_from_solution(students_positions[group*3]))
    st2=SAT_lib.BV_to_number(s.get_BV_from_solution(students_positions[group*3+1]))
    st3=SAT_lib.BV_to_number(s.get_BV_from_solution(students_positions[group*3+2]))

    print ("students:", st1, st2, st3)

    c12=POPCNT(interests[st1]&interests[st2])
    c13=POPCNT(interests[st1]&interests[st3])
    c23=POPCNT(interests[st2]&interests[st3])

    print ("common interests between 1 and 2:", c12)
    print ("common interests between 1 and 3:", c13)
    print ("common interests between 2 and 3:", c23)

    total=c12+c13+c23
    print ("total=", total)
    total_in_all_groups=total_in_all_groups+total

print ("* total in all groups=", total_in_all_groups)
```

Most significant parts from the library used, are:

```
    # bitvectors must be different.
    def fix_BV_NEQ(self, l1, l2):
        #print len(l1), len(l2)
        assert len(l1)==len(l2)
        self.add_comment("fix_BV_NEQ")
        t=[self.XOR(l1[i], l2[i]) for i in range(len(l1))]
        self.add_clause(t)

    def make_distinct_BVs (self, lst):
        assert type(lst)==list
        assert type(lst[0])==list
        for pair in itertools.combinations(lst, r=2):
            self.fix_BV_NEQ(pair[0], pair[1])

...

    def create_MUX(self, ins, sels):
        assert 2**len(sels)==len(ins)
        x=self.create_var()
        for sel in range(len(ins)): # for example, 32 for 5-bit selector
            tmp=[self.neg_if((sel>>i)&1==1, sels[i]) for i in range(len(sels))] # 5
               for 5-bit selector

            self.add_clause([self.neg(ins[sel])] + tmp + [x])
            self.add_clause([ins[sel]] + tmp + [self.neg(x)])
        return x

    # for 1-bit sel
    # ins=[[outputs for sel==0], [outputs for sel==1]]
    def create_wide_MUX (self, ins, sels):
        out=[]
        for i in range(len(ins[0])):
            inputs=[x[i] for x in ins]
            out.append(self.create_MUX(inputs, sels))
        return out
```

(https://smt.st/current_tree/libs/SAT_lib.py)

... and then Open-WBO MaxSAT searches such a solution, for which as many soft clauses as possible would be satisfied, i.e., as many hobbies shared, as possible.

And the result:

```
* group 0
students: 7 12 1
common interests between 1 and 2: 1
common interests between 1 and 3: 1
common interests between 2 and 3: 0
total= 2
* group 1
students: 13 2 10
common interests between 1 and 2: 0
common interests between 1 and 3: 1
common interests between 2 and 3: 0
total= 1
* group 2
students: 3 4 14
common interests between 1 and 2: 1
common interests between 1 and 3: 1
common interests between 2 and 3: 0
total= 2
* group 3
students: 8 5 11
common interests between 1 and 2: 0
common interests between 1 and 3: 0
common interests between 2 and 3: 1
total= 1
* group 4
students: 9 0 6
common interests between 1 and 2: 1
common interests between 1 and 3: 1
common interests between 2 and 3: 1
total= 3
* total between all groups= 9
```

Surely, you can group any other objects with each other based on multiple preferences.

14.9 Finding optimal beer can size using SMT solver

This is classic calculus problem: given a volume of a can, find such a height and radius of a can, so that you'll spent least material to make it (tin, or whatever you use).

Since my toy SMT-solver MK85 supports only integers (in bitvectors) instead of reals, I can try to solve this problem on integers. What to do with π? Let's round it to 3.

```
(declare-fun AlmostPi () (_ BitVec 16))
(assert (= AlmostPi (_ bv3 16)))

; unknowns
(declare-fun Radius () (_ BitVec 16))
(declare-fun Height () (_ BitVec 16))

; There are may not be solutions for Volume=10000, since this we work on integers
; So let's ask for solution for Volume somewhere in between 10000 and 10500...
```

```
(declare-fun Volume () (_ BitVec 16))
(assert (bvuge Volume (_ bv10000 16)))
(assert (bvule Volume (_ bv10500 16)))

; r*r
(declare-fun Radius2 () (_ BitVec 16))
(assert (= Radius2 (bvmul_no_overflow Radius Radius)))

; pi*r*r
(declare-fun AreaOfBase () (_ BitVec 16))
(assert (= AreaOfBase (bvmul_no_overflow Radius2 AlmostPi)))

; 2*pi*r
(declare-fun Circumference () (_ BitVec 16))
(assert (= Circumference (bvmul_no_overflow (bvmul_no_overflow Radius AlmostPi) (_
   bv2 16))))

; Volume = Height * AreaOfBase
(assert (= Volume (bvmul_no_overflow Height AreaOfBase)))

; surface of cylinder = Circumference * Height + 2*AreaOfBase
(declare-fun SurfaceOfCylinder () (_ BitVec 16))
(assert (= SurfaceOfCylinder (bvadd (bvmul_no_overflow Circumference Height) (
   bvmul_no_overflow AreaOfBase (_ bv2 16)))))

(minimize SurfaceOfCylinder)

(check-sat)
(get-model)
```

The solution:

```
sat
(model
        (define-fun AlmostPi () (_ BitVec 16) (_ bv3 16)) ; 0x3
        (define-fun AreaOfBase () (_ BitVec 16) (_ bv507 16)) ; 0x1fb
        (define-fun Circumference () (_ BitVec 16) (_ bv78 16)) ; 0x4e
        (define-fun Height () (_ BitVec 16) (_ bv20 16)) ; 0x14
        (define-fun Radius () (_ BitVec 16) (_ bv13 16)) ; 0xd
        (define-fun Radius2 () (_ BitVec 16) (_ bv169 16)) ; 0xa9
        (define-fun SurfaceOfCylinder () (_ BitVec 16) (_ bv2574 16)) ; 0xa0e
        (define-fun Volume () (_ BitVec 16) (_ bv10140 16)) ; 0x279c
)
```

For Volume=10140, Radius=13, Height=20. Hard to believe, but this is almost correct. I can check it with Wolfram Mathematica:

```
In[]:= FindMinimum[{2*Pi*r*h + Pi*r*r*2, h*Pi*r*r == 10000 }, {r, h}]
Out[]= {2569.5, {r -> 11.6754, h -> 23.3509}}
```

Error is 3.

Using my toy solver and π fixed to 3, perhaps, my results cannot be used in practice, however, we can deduce a general rule: height must be the same as diameter, so that you'll spent minimum tin (or other material) to make it.

Probably, the can will not be suitable for stacking, packing, transporting, or holding in hand, but this is the most economical way to produce them.

14.9.1 A jar

What about a jar (a can without top (or bottom))?

```
...
; surface of jar = Circumference * Height + AreaOfBase
(declare-fun SurfaceOfJar () (_ BitVec 16))
```

```
(assert (= SurfaceOfJar (bvadd (bvmul_no_overflow Circumference Height) AreaOfBase)))

(minimize SurfaceOfJar)
...
```

```
sat
(model
        (define-fun AlmostPi () (_ BitVec 16) (_ bv3 16)) ; 0x3
        (define-fun AreaOfBase () (_ BitVec 16) (_ bv675 16)) ; 0x2a3
        (define-fun Circumference () (_ BitVec 16) (_ bv90 16)) ; 0x5a
        (define-fun Height () (_ BitVec 16) (_ bv15 16)) ; 0xf
        (define-fun Radius () (_ BitVec 16) (_ bv15 16)) ; 0xf
        (define-fun Radius2 () (_ BitVec 16) (_ bv225 16)) ; 0xe1
        (define-fun SurfaceOfJar () (_ BitVec 16) (_ bv2025 16)) ; 0x7e9
        (define-fun Volume () (_ BitVec 16) (_ bv10125 16)) ; 0x278d
)
```

For Volume=10125, Radius=15 and Height=15. We can see that for jar, the height must be equal to the radius. Let's recheck in Mathematica:

```
In[]:= FindMinimum[{2*Pi*r*h + Pi*r*r, h*Pi*r*r == 10000 }, {r, h}]
Out[]= {2039.41, {r -> 14.7101, h -> 14.7101}}
```

Correct!

Yes, you've been probably taught in school to solve this using paper and pencil, but... The fun thing is that I never knew calculus at all, but I could write my toy bit-blaster, which can give such answers. And thanks to Open-WBO, which is used in my MK85 as external MaxSAT solver.

Since I'm using non-standard SMT-LIB function `bvmul_no_overflow`, this will not work on other SMT solvers. For Z3, `bvumul_noovfl` is to be used: https://github.com/Z3Prover/z3/issues/574

See also: https://demonstrations.wolfram.com/MinimizingTheSurfaceAreaOfACylinderWithAFixedVolume/

14.10 Choosing between short/long jumps in x86 assembler

As you may know, there are two JMP instructions in x86: short one (EB xx) and long one (E9 xx xx xx xx). The first can encode *short offsets*: [current_address-127 ... current_address+128], the second can encode 32-bit offset. During assembling (converting assembly code into machine opcodes) you can put *long* JMPs, and it's OK. But here is a problem: you may want to make your code as tight as possible and use *short* JMPs whenever possible. Given the fact that JMPs are inside code itself and affecting code size. What can you do?

This is an example of some assembly program:

```
label_1:
         ;
         ; block 1     block1_size
         ;
         JMP label_3    JMP_1_size
label_2:
         ;
         ; block 2     block2_size
         ;
         JMP label_5    JMP_2_size
label_3:
         ;
         ; block 3     block3_size
         ;
         JMP label_2    JMP_3_size
label_4:
         ;
         ; block 4     block4_size
         ;
         JMP label_1    JMP_4_size
label_5:
         ;
         ; block 5      block5_size
         ;
         JMP label_3    JMP_5_size
```

```
from z3 import *

# this is simplification, "back" offsets are limited by 127 bytes, "forward" ones by
   128 bytes, but OK, let's say,
# all of them are 128:
def JMP_size (offset):
    return If(offset>128, 5, 2)

block1_size=64
block2_size=81
block3_size=12
block4_size=50
block5_size=60

s=Optimize()

JMP_1_size=Int('JMP_1_size')
JMP_2_size=Int('JMP_2_size')
JMP_3_size=Int('JMP_3_size')
JMP_4_size=Int('JMP_4_size')
JMP_5_size=Int('JMP_5_size')

JMP_1_offset=Int('JMP_1_offset')
JMP_2_offset=Int('JMP_2_offset')
JMP_3_offset=Int('JMP_3_offset')
JMP_4_offset=Int('JMP_4_offset')
JMP_5_offset=Int('JMP_5_offset')

# calculate all JMPs offsets, these are block sizes and also other's JMPs sizes
   between the current address
# and destination address:
s.add(JMP_1_offset==block2_size+JMP_2_size)
s.add(JMP_2_offset==block3_size+JMP_3_size + block4_size+JMP_4_size)
s.add(JMP_3_offset==block2_size+JMP_2_size + block3_size)
s.add(JMP_4_offset==block1_size+JMP_1_size + block2_size+JMP_2_size + block3_size+
   JMP_3_size + block4_size)
s.add(JMP_5_offset==block3_size+JMP_3_size + block4_size+JMP_4_size + block5_size)

# what are sizes of all JMPs, 2 or 5?
s.add(JMP_1_size==JMP_size(JMP_1_offset))
s.add(JMP_2_size==JMP_size(JMP_2_offset))
s.add(JMP_3_size==JMP_size(JMP_3_offset))
s.add(JMP_4_size==JMP_size(JMP_4_offset))
s.add(JMP_5_size==JMP_size(JMP_5_offset))

# minimize size of all jumps (this is optimization problem):
s.minimize(JMP_1_size + JMP_2_size + JMP_3_size + JMP_4_size + JMP_5_size)

print s.check()
print s.model()
```

Listing 14.2: The result

```
sat
[JMP_2_size = 2,
 JMP_1_offset = 83,
 JMP_5_size = 5,
 JMP_5_offset = 129,
 JMP_2_offset = 69,
 JMP_4_size = 5,
 JMP_4_offset = 213,
 JMP_1_size = 2,
 JMP_3_size = 2,
 JMP_3_offset = 95]
```

I.e., JMP_4 and JMP_5 JMPs must be *long* ones, others can be *short* ones.

Other simplification I made for the sake of example: *short* conditional Jcc's can also be encoded using 2 bytes, *long* ones using 6 bytes rather than 5 (5 is unconditional JMP).

Chapter 15

Synthesis

Synthesis is construction of some structure, according to specification...

15.1 Logic circuits synthesis

15.1.1 Simple logic synthesis using Z3 and Apollo Guidance Computer

What a smallest possible logic circuit is possible for a given truth table?

Let's try to define truth tables for inputs and outputs and find smallest circuit. This program is almost the same as I mentioned earlier: 15.2.1, but reworked slightly:

```
#!/usr/bin/env python3

from z3 import *
import sys

I_AND, I_OR, I_XOR, I_NOT, I_NOR3 = 0,1,2,3,4

# 1-bit NOT
"""
INPUTS=[0b10]
OUTPUTS=[0b01]
BITS=2
add_always_false=False
add_always_true=True
avail=[I_XOR]
#avail=[I_NOR3]
"""

# 2-input AND
"""
INPUTS=[0b1100, 0b1010]
OUTPUTS=[0b1000]
BITS=4
add_always_false=False
add_always_true=False
avail=[I_OR, I_NOT]
#avail=[I_NOR3]
"""

# 2-input XOR
#"""
INPUTS=[0b1100, 0b1010]
OUTPUTS=[0b0110]
BITS=4
add_always_false=False
#add_always_false=True
```

```
add_always_true=False
#add_always_true=True
#avail=[I_NOR3]
#avail=[I_AND, I_NOT]
avail=[I_OR, I_NOT]
#"""

# parity (or popcnt1)
""" TT for parity
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 0
"""
"""
INPUTS=[0b11110000, 0b11001100, 0b10101010]
OUTPUTS=[0b00010110]
BITS=8
add_always_false=False
add_always_true=False
#add_always_false=True
#add_always_true=True
avail=[I_AND, I_XOR, I_OR, I_NOT]
#avail=[I_XOR, I_OR, I_NOT]
#avail=[I_AND, I_OR, I_NOT]
"""

# full-adder
"""
INPUTS=[0b11110000, 0b11001100, 0b10101010]
OUTPUTS=[0b11101000, 0b10010110] # carry-out, sum
BITS=8
add_always_false=False
add_always_true=False
avail=[I_AND, I_OR, I_XOR, I_NOT]
#avail=[I_AND, I_OR, I_NOT]
#avail=[I_NOR3]
"""

# popcnt
""" TT for popcnt

in  HL
000 00
001 01
010 01
011 10
100 01
101 10
110 10
111 11
"""
"""
INPUTS=[0b11110000, 0b11001100, 0b10101010]
OUTPUTS=[0b11101000, 0b10010110] # high, low
BITS=8
add_always_false=False
```

```
add_always_true=False
#avail=[I_AND, I_OR, I_XOR, I_NOT]
avail=[I_NOR3]
"""

# majority for 3 bits
""" TT for majority (true if 2 or 3 bits are True)
000 0
001 0
010 0
011 1
100 0
101 1
110 1
111 1
"""
"""
INPUTS=[0b11110000, 0b11001100, 0b10101010]
OUTPUTS=[0b11101000]
BITS=8
add_always_false=False
add_always_true=False
avail=[I_AND, I_OR, I_XOR, I_NOT]
"""

# 2-bit adder:
"""
INPUTS=[0b1111111100000000, 0b1111000011110000, 0b1100110011001100, 0
    b1010101010101010]
OUTPUTS=[0b1001001101101100, 0b0101101001011010] # high, low
BITS=16
add_always_false=True
add_always_true=True
#add_always_false=False
#add_always_true=False
avail=[I_AND, I_OR, I_XOR, I_NOT]
#avail=[I_NOR3]
"""

"""
#7-segment display
INPUTS=[0b1111111100000000, 0b1111000011110000, 0b1100110011001100, 0
    b1010101010101010]
# "g" segment, like here: http://www.nutsvolts.com/uploads/wygwam/
    NV_0501_Marston_Figure02.jpg
OUTPUTS=[0b1110111101111100] # g
BITS=16
add_always_false=False
add_always_true=False
avail=[I_AND, I_OR, I_XOR, I_NOT]
#avail=[I_NOR3]
"""

MAX_STEPS=20

# if additional always-false or always-true must be present:
if add_always_false:
    INPUTS.append(0)
if add_always_true:
    INPUTS.append(2**BITS-1)
```

```
# this called during self-testing:
def eval_ins(R, s, m, STEPS, op, op1_reg, op2_reg, op3_reg):
    op_n=m[op[s]].as_long()
    op1_reg_tmp=m[op1_reg[s]].as_long()
    op1_val=R[op1_reg_tmp]
    op2_reg_tmp=m[op2_reg[s]].as_long()
    op3_reg_tmp=m[op3_reg[s]].as_long()
    if op_n in [I_AND, I_OR, I_XOR, I_NOR3]:
        op2_val=R[op2_reg_tmp]
    if op_n==I_AND:
        return op1_val&op2_val

    elif op_n==I_OR:
        return op1_val|op2_val

    elif op_n==I_XOR:
        return op1_val^op2_val

    elif op_n==I_NOT:
        return ~op1_val

    elif op_n==I_NOR3:
        op3_val=R[op3_reg_tmp]
        return ~(op1_val|op2_val|op3_val)

    else:
        raise AssertionError

# evaluate program we've got. for self-testing.
def eval_pgm(m, STEPS, op, op1_reg, op2_reg, op3_reg):
    R=[None]*STEPS
    for i in range(len(INPUTS)):
        R[i]=INPUTS[i]

    for s in range(len(INPUTS),STEPS):
        R[s]=eval_ins(R, s, m, STEPS, op, op1_reg, op2_reg, op3_reg)

    return R

# get all states, for self-testing:
def selftest(m, STEPS, op, op1_reg, op2_reg, op3_reg):
    l=eval_pgm(m, STEPS, op, op1_reg, op2_reg, op3_reg)
    print ("simulate:")
    for i in range(len(l)):
        print ("r%d=" % i, format(l[i] & 2**BITS-1, '0'+str(BITS)+'b'))

"""
selector() functions generates expression like:

If(op1_reg_s5 == 0,
   S_s0,
   If(op1_reg_s5 == 1,
      S_s1,
      If(op1_reg_s5 == 2,
         S_s2,
         If(op1_reg_s5 == 3,
            S_s3,
            If(op1_reg_s5 == 4,
               S_s4,
               If(op1_reg_s5 == 5,
                  S_s5,
                  If(op1_reg_s5 == 6,
                     S_s6,
                     If(op1_reg_s5 == 7,
                        S_s7,
                        If(op1_reg_s5 == 8,
                           S_s8,
                           If(op1_reg_s5 == 9,
                              S_s9,
                              If(op1_reg_s5 == 10,
                                 S_s10,
                                 If(op1_reg_s5 == 11,
                                    S_s11,
                                    0))))))))))))

this is like multiplexer or switch()
"""
def selector(R, s):
    t=0 # default value
    for i in range(MAX_STEPS):
        t=If(s==(MAX_STEPS-i-1), R[MAX_STEPS-i-1], t)
    return t

def simulate_op(R, op, op1_reg, op2_reg, op3_reg):
    op1_val=selector(R, op1_reg)
    return If(op==I_AND, op1_val & selector(R, op2_reg),
           If(op==I_OR, op1_val | selector(R, op2_reg),
           If(op==I_XOR, op1_val ^ selector(R, op2_reg),
           If(op==I_NOT, ~op1_val,
           If(op==I_NOR3, ~(op1_val | selector(R, op2_reg) | selector (R, op3_reg)),
               0))))) # default

op_to_str_tbl=["AND", "OR", "XOR", "NOT", "NOR3"]

def print_model(m, R, STEPS, op, op1_reg, op2_reg, op3_reg):
    print ("%d instructions" % (STEPS-len(INPUTS)))
    for s in range(STEPS):
        if s<len(INPUTS):
            t="r%d=input" % s
        else:
            op_n=m[op[s]].as_long()
            op_s=op_to_str_tbl[op_n]
            op1_reg_n=m[op1_reg[s]].as_long()
            op2_reg_n=m[op2_reg[s]].as_long()
            op3_reg_n=m[op3_reg[s]].as_long()
            if op_n in [I_AND, I_OR, I_XOR]:
                t="r%d=%s r%d, r%d" % (s, op_s, op1_reg_n, op2_reg_n)
            elif op_n==I_NOT:
                t="r%d=%s r%d" % (s, op_s, op1_reg_n)
            else: # else NOR3
                t="r%d=%s r%d, r%d, r%d" % (s, op_s, op1_reg_n, op2_reg_n, op3_reg_n)

        tt=format(m[R[s]].as_long(), '0'+str(BITS)+'b')
        print (t+" "*(25-len(t))+tt)

def attempt(STEPS):
    print ("attempt, STEPS=", STEPS)
    sl=Solver()

    # state of each register:
    R=[BitVec(('S_s%d' % s), BITS) for s in range(MAX_STEPS)]
    # operation type and operands for each register:
```

```
    op=[Int('op_s%d' % s) for s in range(MAX_STEPS)]
    op1_reg=[Int('op1_reg_s%d' % s) for s in range(MAX_STEPS)]
    op2_reg=[Int('op2_reg_s%d' % s) for s in range(MAX_STEPS)]
    op3_reg=[Int('op3_reg_s%d' % s) for s in range(MAX_STEPS)]

    for s in range(len(INPUTS), STEPS):
        # for each step.
        # expression like Or(op[s]==0, op[s]==1, ...) is formed here. values are
           taken from avail[]
        sl.add(Or(*[op[s]==j for j in avail]))
        # each operand can refer to one of registers BEFORE the current instruction:
        sl.add(And(op1_reg[s]>=0, op1_reg[s]<s))
        sl.add(And(op2_reg[s]>=0, op2_reg[s]<s))
        sl.add(And(op3_reg[s]>=0, op3_reg[s]<s))

    # fill inputs:
    for i in range(len(INPUTS)):
        sl.add(R[i]==INPUTS[i])
    # fill outputs, "must be's"
    for o in range(len(OUTPUTS)):
        sl.add(R[STEPS-(o+1)]==list(reversed(OUTPUTS))[o])

    # connect each register to "simulator":
    for s in range(len(INPUTS), STEPS):
        sl.add(R[s]==simulate_op(R, op[s], op1_reg[s], op2_reg[s], op3_reg[s]))

    tmp=sl.check()
    if tmp==sat:
        print ("sat!")
        m=sl.model()
        #print (m)
        print_model(m, R, STEPS, op, op1_reg, op2_reg, op3_reg)
        selftest(m, STEPS, op, op1_reg, op2_reg, op3_reg)
        exit(0)
    else:
        print (tmp)

for s in range(len(INPUTS)+len(OUTPUTS), MAX_STEPS):
    attempt(s)
```

I could generate only small circuits maybe up to ≈ 10 gates, but this is interesting nonetheless.

Also, I've always wondering how you can do something usable for Apollo Guidance Computer, which had only one single gate: NOR3? See also its schematics: http://klabs.org/history/ech/agc_schematics/. The answer is De Morgan's laws, but this is not obvious.

INPUTS[] has all possible bit combinations for all inputs, or all possible truth tables. OUTPUTS[] has truth table for each output. All the rest is processed in bitsliced manner. Given that, the resulting program will work on 4/8/16-bit CPU and will generate defined OUTPUTS for defined INPUTS. Or, this program can be treated just like a logic circuit.

AND gate

How to build 2-input AND gate using only OR's and NOT's?

```
INPUTS=[0b1100, 0b1010]
OUTPUTS=[0b1000]
BITS=4
avail=[I_OR, I_NOT]

...

r0=input                 1100
r1=input                 1010
```

```
r2=NOT r1                 0101
r3=NOT r0                 0011
r4=OR r3, r2              0111
r5=NOT r4                 1000
```

This is indeed like stated in De Morgan's laws: $x \wedge y$ is equivalent to $\neg(\neg x \vee \neg y)$. Can be used for obfuscation? Now using only NOR3 gate?

```
avail=[I_NOR3]

...

r0=input                  1100
r1=input                  1010
r2=NOR3 r1, r1, r1        0101
r3=NOR3 r2, r0, r0        0010
r4=NOR3 r3, r2, r2        1000
```

XOR gate

How to build 2-input XOR using only OR's and NOT's?

```
INPUTS=[0b1100, 0b1010]
OUTPUTS=[0b0110]
BITS=4
avail=[I_OR, I_NOT]

...

7 instructions
r0=input                  1100
r1=input                  1010
r2=OR r1, r0              1110
r3=NOT r2                 0001
r4=OR r0, r3              1101
r5=NOT r4                 0010
r6=OR r1, r3              1011
r7=NOT r6                 0100
r8=OR r5, r7              0110
```

... using only AND's and NOT's?

```
avail=[I_AND, I_NOT]
...
r0=input                  1100
r1=input                  1010
r2=NOT r1                 0101
r3=AND r1, r0             1000
r4=NOT r3                 0111
r5=NOT r0                 0011
r6=AND r2, r5             0001
r7=NOT r6                 1110
r8=AND r4, r7             0110
```

... using only NOR3 gates?

```
avail=[I_NOR3]
...
r0=input                  1100
r1=input                  1010
r2=NOR3 r1, r1, r1        0101
r3=NOR3 r0, r0, r1        0001
r4=NOR3 r0, r2, r3        0010
```

```
r5=NOR3 r2, r4, r2        1000
r6=NOR3 r3, r5, r3        0110
```

Full-adder

According to Wikipedia, full-adder can be constructed using two XOR gates, two AND gates and one OR gate. But I had no idea 3 XORs and 2 ANDs can be used instead:

```
INPUTS=[0b11110000, 0b11001100, 0b10101010]
OUTPUTS=[0b11101000, 0b10010110] # carry-out, sum
BITS=8
avail=[I_AND, I_OR, I_XOR, I_NOT]

...

5 instructions
r0=input                  11110000
r1=input                  11001100
r2=input                  10101010
r3=XOR r2, r1             01100110
r4=AND r0, r3             01100000
r5=AND r2, r1             10001000
r6=XOR r4, r5             11101000
r7=XOR r0, r3             10010110
```

... using only NOR3 gates:

```
avail=[I_NOR3]
...
8 instructions
r0=input                  11110000
r1=input                  11001100
r2=input                  10101010
r3=NOR3 r0, r0, r1        00000011
r4=NOR3 r2, r3, r1        00010000
r5=NOR3 r3, r2, r0        00000100
r6=NOR3 r3, r0, r5        00001000
r7=NOR3 r5, r2, r4        01000001
r8=NOR3 r3, r4, r1        00100000
r9=NOR3 r3, r5, r4        11101000
r10=NOR3 r8, r7, r6       10010110
```

POPCNT

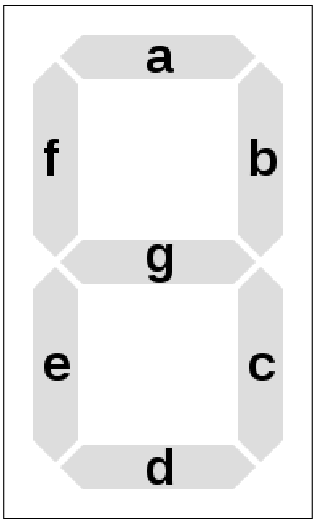
Smallest circuit to count bits in 3-bit input, producing 2-bit output:

```
INPUTS=[0b11110000, 0b11001100, 0b10101010]
OUTPUTS=[0b11101000, 0b10010110] # high, low
BITS=8
avail=[I_AND, I_OR, I_XOR, I_NOT]
...
5 instructions
r0=input                  11110000
r1=input                  11001100
r2=input                  10101010
r3=XOR r2, r1             01100110
r4=AND r0, r3             01100000
r5=AND r2, r1             10001000
r6=XOR r4, r5             11101000
r7=XOR r0, r3             10010110
```

... using only NOR3 gates:

```
avail=[I_NOR3]
...
8 instructions
r0=input                 11110000
r1=input                 11001100
r2=input                 10101010
r3=NOR3 r0, r0, r1       00000011
r4=NOR3 r2, r3, r1       00010000
r5=NOR3 r3, r2, r0       00000100
r6=NOR3 r3, r0, r5       00001000
r7=NOR3 r5, r2, r4       01000001
r8=NOR3 r3, r4, r1       00100000
r9=NOR3 r3, r5, r4       11101000
r10=NOR3 r8, r7, r6      10010110
```

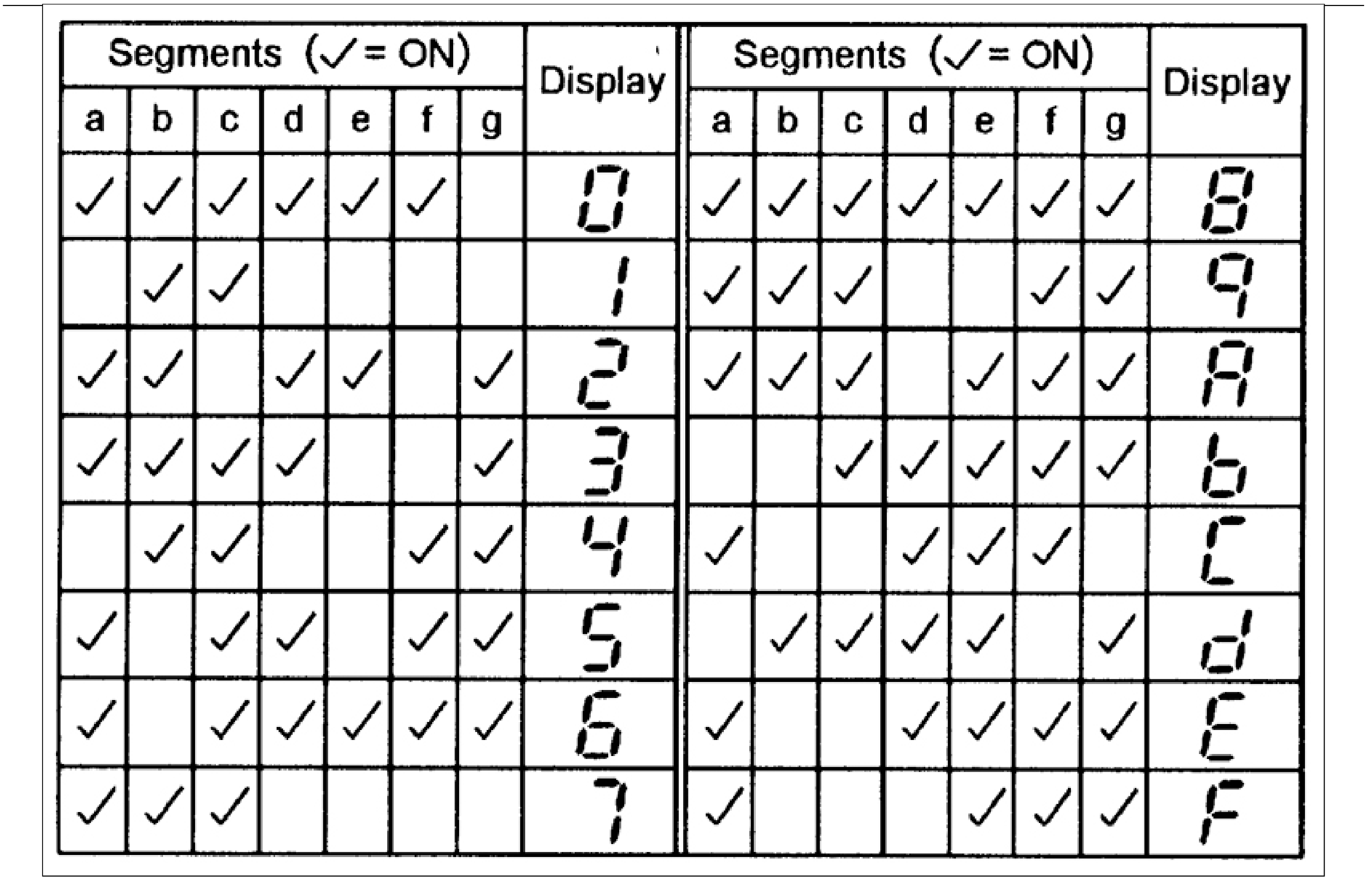
Circuit for a central ”g” segment of 7-segment display



(The image taken from Wikipedia.)

I couldn’t find a circuit for the all 7 segments, but found for one, a central one (”g”). Yes, encoders like these are usually implemented using a ROM. But I always been wondering, how to do this using only gates.

The truth table for ”g” segment I’ve used from this table:



Segments (✓= ON)							Display	Segments (✓= ON)							Display
a	b	c	d	e	f	g		a	b	c	d	e	f	g	
✓	✓	✓	✓	✓	✓		0	✓	✓	✓	✓	✓	✓	✓	8
	✓	✓					1	✓	✓	✓			✓	✓	9
✓	✓		✓	✓		✓	2	✓	✓	✓		✓	✓	✓	A
✓	✓	✓	✓			✓	3			✓	✓	✓	✓	✓	b
	✓	✓			✓	✓	4	✓			✓	✓	✓		C
✓		✓	✓		✓	✓	5		✓	✓	✓	✓		✓	d
✓		✓	✓	✓	✓	✓	6	✓			✓	✓	✓	✓	E
✓	✓	✓					7	✓				✓	✓	✓	F

(The image taken from http://www.nutsvolts.com.)

```
INPUTS=[0b1111111100000000, 0b1111000011110000, 0b1100110011001100, 0
    b1010101010101010]
# "g" segment, like here: http://www.nutsvolts.com/uploads/wygwam/
    NV_0501_Marston_Figure02.jpg
OUTPUTS=[0b1110111101111100] # g
BITS=16
avail=[I_AND, I_OR, I_XOR, I_NOT]
...
5 instructions
r0=input                1111111100000000
r1=input                1111000011110000
r2=input                1100110011001100
r3=input                1010101010101010
r4=AND r3, r1           1010000010100000
r5=XOR r4, r0           0101111110100000
r6=XOR r4, r2           0110110001101100
r7=XOR r5, r1           1010111101010000
r8=OR r7, r6            1110111101111100
```

Using only NOR3 gates:

```
avail=[I_NOR3]
...
8 instructions
r0=input                1111111100000000
r1=input                1111000011110000
r2=input                1100110011001100
r3=input                1010101010101010
r4=NOR3 r1, r1, r1      0000111100001111
r5=NOR3 r3, r3, r4      0101000001010000
r6=NOR3 r2, r0, r0      0000000000110011
r7=NOR3 r6, r6, r5      1010111110001100
```

```
r8=NOR3 r4, r0, r7       0000000001110000
r9=NOR3 r8, r7, r2       0001000000000011
r10=NOR3 r0, r8, r4      0000000010000000
r11=NOR3 r9, r10, r10    1110111101111100
```

But what about bruteforce?

Let's imagine, a program of 5 instructions, each can be one of four (AND/OR/XOR/NOT).

There are can be $4^2 \cdot 4$ first instructions (that is connected to r4). Two operands, each can be connected to one of 4 inputs (r0...r3). Four instructions: 4.

There are can be $5^2 \cdot 4$ second instructions... and $8^2 \cdot 4$ fifth instructions.

How many programs there can be?

$$\prod_{j=i}^{s+i-1} j^o \cdot m$$

Figure 15.1: The general formula. j = how many inputs available to an instruction at a step; i = total number of inputs of a program (4 in my case); s = total number of instructions in program (*steps*); o = total number of operands each instruction has; m = how many instructions available to choose from.

$(4^2 \cdot 4) \cdot (5^2 \cdot 4) \cdot (6^2 \cdot 4) \cdot (7^2 \cdot 4) \cdot (8^2 \cdot 4) = 46242201600$

And $log_2(46242201600) \approx 35$ bits. This is feasible on a fast computer.

Now what about a program of 8 instructions, where each instruction is always NOR and has 3 inputs/operands? 1st instruction: 4^3 ... 8th instruction: 11^3.

How many programs there can be?

$4^3 \cdot 5^3 \cdot 6^3 \cdot 7^3 \cdot 8^3 \cdot 9^3 \cdot 10^3 \cdot 11^3 = 294451250429952000000$. And $log_2(294451250429952000000) \approx 68$ bits. Way too much for bruteforce.

15.1.2 TAOCP 7.1.1 exercises 4 and 5

Found this exercise in TAOCP section 7.1.1 (Boolean basics):

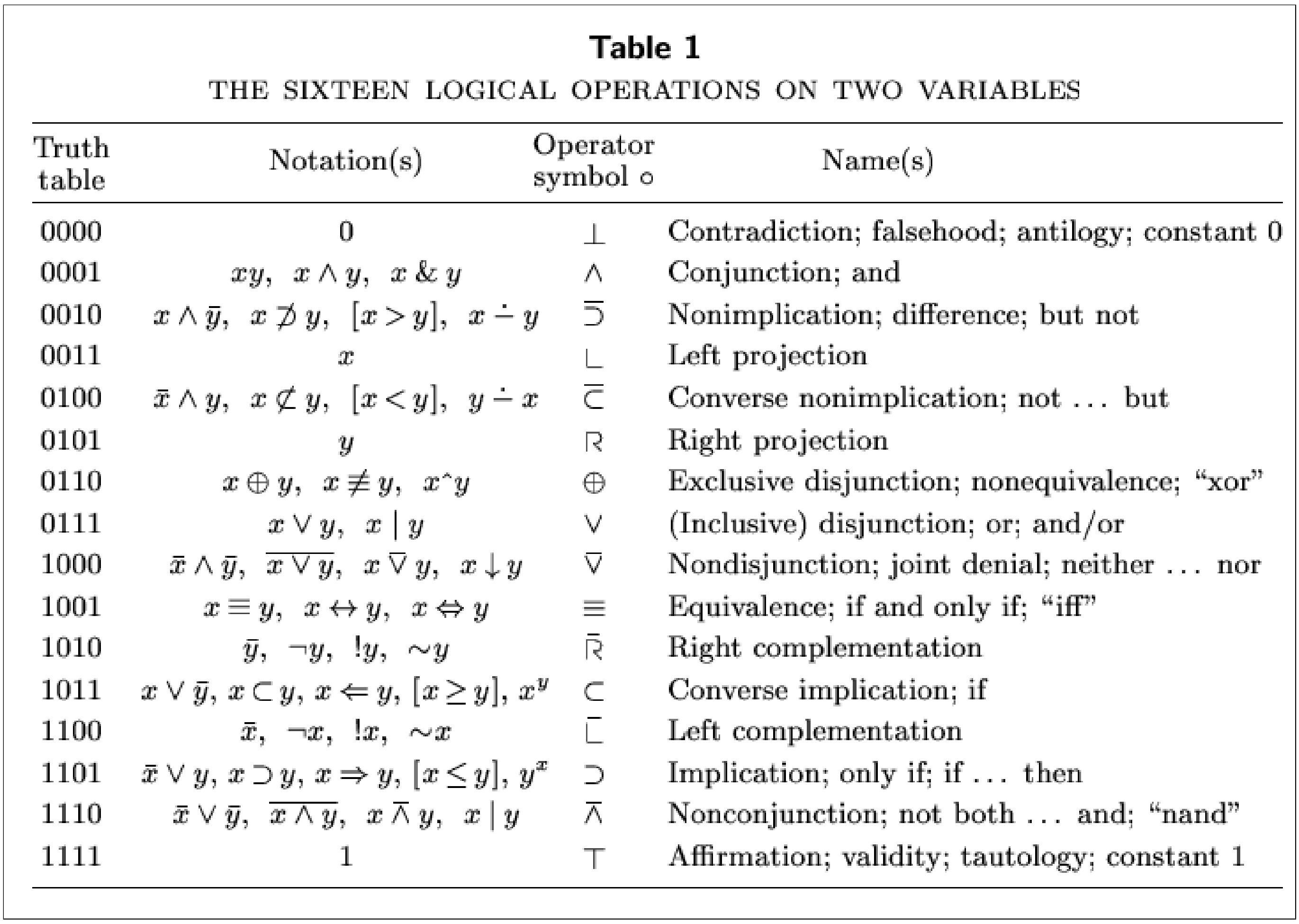
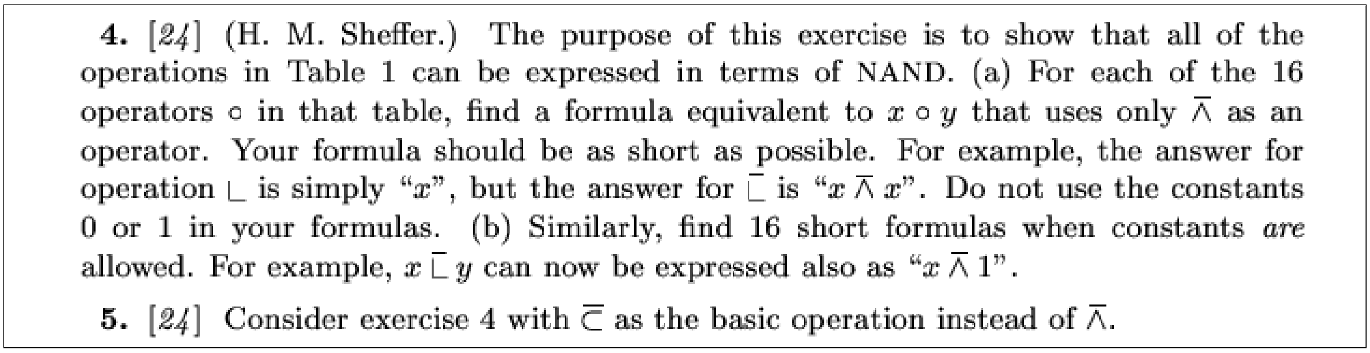


Table 1

THE SIXTEEN LOGICAL OPERATIONS ON TWO VARIABLES

Truth table	Notation(s)	Operator symbol $\circ$	Name(s)
0000	0	$\bot$	Contradiction; falsehood; antilogy; constant 0
0001	$xy,\ x\wedge y,\ x \,\&\, y$	$\wedge$	Conjunction; and
0010	$x\wedge\bar{y},\ x\not\supset y,\ [x>y],\ x\dot{-}y$	$\bar{\supset}$	Nonimplication; difference; but not
0011	x	$\llcorner$	Left projection
0100	$\bar{x}\wedge y,\ x\not\subset y,\ [x<y],\ y\dot{-}x$	$\bar{\subset}$	Converse nonimplication; not ... but
0101	y	R	Right projection
0110	$x\oplus y,\ x\not\equiv y,\ x\hat{\ }y$	$\oplus$	Exclusive disjunction; nonequivalence; “xor”
0111	$x\vee y,\ x\mid y$	$\vee$	(Inclusive) disjunction; or; and/or
1000	$\bar{x}\wedge\bar{y},\ \overline{x\vee y},\ x\,\bar{\vee}\,y,\ x\downarrow y$	$\bar{\vee}$	Nondisjunction; joint denial; neither ... nor
1001	$x\equiv y,\ x\leftrightarrow y,\ x\Leftrightarrow y$	$\equiv$	Equivalence; if and only if; “iff”
1010	$\bar{y},\ \neg y,\ !y,\ \sim y$	$\bar{\mathsf{R}}$	Right complementation
1011	$x\vee\bar{y}, x\subset y, x\Leftarrow y, [x\geq y], x^y$	$\subset$	Converse implication; if
1100	$\bar{x},\ \neg x,\ !x,\ \sim x$	$\bar{\llcorner}$	Left complementation
1101	$\bar{x}\vee y, x\supset y, x\Rightarrow y, [x\leq y], y^x$	$\supset$	Implication; only if; if ... then
1110	$\bar{x}\vee\bar{y},\ \overline{x\wedge y},\ x\,\bar{\wedge}\,y,\ x\mid y$	$\bar{\wedge}$	Nonconjunction; not both ... and; “nand”
1111	1	$\top$	Affirmation; validity; tautology; constant 1

Figure 15.2: Page 3



4. [*24*] (H. M. Sheffer.) The purpose of this exercise is to show that all of the operations in Table 1 can be expressed in terms of NAND. (a) For each of the 16 operators $\circ$ in that table, find a formula equivalent to $x\circ y$ that uses only $\bar{\wedge}$ as an operator. Your formula should be as short as possible. For example, the answer for operation $\llcorner$ is simply “x”, but the answer for $\bar{\llcorner}$ is “$x\,\bar{\wedge}\,x$”. Do not use the constants 0 or 1 in your formulas. (b) Similarly, find 16 short formulas when constants *are* allowed. For example, $x\,\bar{\llcorner}\,y$ can now be expressed also as “$x\,\bar{\wedge}\,1$”.

5. [*24*] Consider exercise 4 with $\bar{\subset}$ as the basic operation instead of $\bar{\wedge}$.

Figure 15.3: Page 34

I’m not clever enough to solve this manually (yet), but I could try using logic synthesis, as I did before. As they say, “machines should work; people should think”.

The modified Z3Py script:

```
#!/usr/bin/env python3

from z3 import *
import sys

I_AND, I_OR, I_XOR, I_NOT, I_NOR3, I_NAND, I_ANDN = 0,1,2,3,4,5,6

# 2-input function
BITS=4
add_always_false=False
```

```
add_always_true=False
#add_always_false=True
#add_always_true=True
avail=[I_NAND]
#avail=[I_ANDN]

MAX_STEPS=10

# My representation of TT is: [MSB..LSB].
# Knuth's representation in the section 7.1.1 (Boolean basics) of TAOCP is different:
    [LSB..MSB].
# so I'll reverse bits before printing TT:
def rvr_4_bits(i):
    return ((i>>0)&1)<<3 | ((i>>1)&1)<<2 | ((i>>2)&1)<<1 | ((i>>3)&1)<<0

def find_NAND_only_for_TT (OUTPUTS):
    INPUTS=[0b1100, 0b1010]
    # if additional always-false or always-true must be present:
    if add_always_false:
        INPUTS.append(0)
    if add_always_true:
        INPUTS.append(2**BITS-1)

    # this called during self-testing:
    def eval_ins(R, s, m, STEPS, op, op1_reg, op2_reg, op3_reg):
        op_n=m[op[s]].as_long()
        op1_reg_tmp=m[op1_reg[s]].as_long()
        op1_val=R[op1_reg_tmp]
        op2_reg_tmp=m[op2_reg[s]].as_long()
        op3_reg_tmp=m[op3_reg[s]].as_long()
        if op_n in [I_AND, I_OR, I_XOR, I_NOR3, I_NAND, I_ANDN]:
            op2_val=R[op2_reg_tmp]
        if op_n==I_AND:
            return op1_val&op2_val

        elif op_n==I_OR:
            return op1_val|op2_val

        elif op_n==I_XOR:
            return op1_val^op2_val

        elif op_n==I_NOT:
            return ~op1_val

        elif op_n==I_NOR3:
            op3_val=R[op3_reg_tmp]
            return ~(op1_val|op2_val|op3_val)
        elif op_n==I_NAND:
            return ~(op1_val&op2_val)
        elif op_n==I_ANDN:
            return (~op1_val)&op2_val

        else:
            raise AssertionError

    # evaluate program we've got. for self-testing.
    def eval_pgm(m, STEPS, op, op1_reg, op2_reg, op3_reg):
        R=[None]*STEPS
        for i in range(len(INPUTS)):
            R[i]=INPUTS[i]
        for s in range(len(INPUTS),STEPS):
            R[s]=eval_ins(R, s, m, STEPS, op, op1_reg, op2_reg, op3_reg)

        return R

    # get all states, for self-testing:
    def selftest(m, STEPS, op, op1_reg, op2_reg, op3_reg):
        l=eval_pgm(m, STEPS, op, op1_reg, op2_reg, op3_reg)
        print ("simulate:")
        for i in range(len(l)):
            print ("r%d=" % i, format(l[i] & 2**BITS-1, '0'+str(BITS)+'b'))

    """
    selector() function generates expression like:

    If(op1_reg_s5 == 0,
       S_s0,
       If(op1_reg_s5 == 1,
          S_s1,
          If(op1_reg_s5 == 2,
             S_s2,
             If(op1_reg_s5 == 3,
                S_s3,
                If(op1_reg_s5 == 4,
                   S_s4,
                   If(op1_reg_s5 == 5,
                      S_s5,
                      If(op1_reg_s5 == 6,
                         S_s6,
                         If(op1_reg_s5 == 7,
                            S_s7,
                            If(op1_reg_s5 == 8,
                               S_s8,
                               If(op1_reg_s5 == 9,
                                  S_s9,
                                  If(op1_reg_s5 == 10,
                                     S_s10,
                                     If(op1_reg_s5 == 11,
                                        S_s11,
                                    0))))))))))))

    this is like multiplexer or switch()
    """
    def selector(R, s):
        t=0 # default value
        for i in range(MAX_STEPS):
            t=If(s==(MAX_STEPS-i-1), R[MAX_STEPS-i-1], t)
        return t

    def simulate_op(R, op, op1_reg, op2_reg, op3_reg):
        op1_val=selector(R, op1_reg)
        return If(op==I_AND, op1_val & selector(R, op2_reg),
               If(op==I_OR, op1_val | selector(R, op2_reg),
               If(op==I_XOR, op1_val ^ selector(R, op2_reg),
               If(op==I_NOT, ~op1_val,
               If(op==I_NOR3, ~(op1_val | selector(R, op2_reg) | selector (R, op3_reg
                  )),
               If(op==I_NAND, ~(op1_val & selector(R, op2_reg)),
               If(op==I_ANDN, (~op1_val) & selector(R, op2_reg),
                   0))))))) # default
```

```
op_to_str_tbl=["AND", "OR", "XOR", "NOT", "NOR3", "NAND", "ANDN"]

def print_model(m, R, STEPS, op, op1_reg, op2_reg, op3_reg):
    print ("%d instructions for OUTPUTS TT (Knuth's representation)=" % (STEPS-len(INPUTS))), format(rvr_4_bits(OUTPUTS[0]) & 2**BITS-1, '0'+str(BITS)+'b')
    for s in range(STEPS):
        if s<len(INPUTS):
            t="r%d=input" % s
        else:
            op_n=m[op[s]].as_long()
            op_s=op_to_str_tbl[op_n]
            op1_reg_n=m[op1_reg[s]].as_long()
            op2_reg_n=m[op2_reg[s]].as_long()
            op3_reg_n=m[op3_reg[s]].as_long()
            if op_n in [I_AND, I_OR, I_XOR, I_NAND, I_ANDN]:
                t="r%d=%s r%d, r%d" % (s, op_s, op1_reg_n, op2_reg_n)
            elif op_n==I_NOT:
                t="r%d=%s r%d" % (s, op_s, op1_reg_n)
            else: # else NOR3
                t="r%d=%s r%d, r%d, r%d" % (s, op_s, op1_reg_n, op2_reg_n, op3_reg_n)

        tt=format(m[R[s]].as_long(), '0'+str(BITS)+'b')
        print (t+" "*(25-len(t))+tt)

def attempt(STEPS):
    print ("attempt, STEPS=", STEPS)
    sl=Solver()

    # state of each register:
    R=[BitVec(('S_s%d' % s), BITS) for s in range(MAX_STEPS)]
    # operation type and operands for each register:
    op=[Int('op_s%d' % s) for s in range(MAX_STEPS)]
    op1_reg=[Int('op1_reg_s%d' % s) for s in range(MAX_STEPS)]
    op2_reg=[Int('op2_reg_s%d' % s) for s in range(MAX_STEPS)]
    op3_reg=[Int('op3_reg_s%d' % s) for s in range(MAX_STEPS)]

    for s in range(len(INPUTS), STEPS):
        # for each step.
        # expression like Or(op[s]==0, op[s]==1, ...) is formed here. values are taken from avail[]
        sl.add(Or(*[op[s]==j for j in avail]))
        # each operand can refer to one of registers BEFORE the current instruction:
        sl.add(And(op1_reg[s]>=0, op1_reg[s]<s))
        sl.add(And(op2_reg[s]>=0, op2_reg[s]<s))
        sl.add(And(op3_reg[s]>=0, op3_reg[s]<s))

    # fill inputs:
    for i in range(len(INPUTS)):
        sl.add(R[i]==INPUTS[i])
    # fill outputs, "must be's"
    for o in range(len(OUTPUTS)):
        sl.add(R[STEPS-(o+1)]==list(reversed(OUTPUTS))[o])

    # connect each register to "simulator":
    for s in range(len(INPUTS), STEPS):
        sl.add(R[s]==simulate_op(R, op[s], op1_reg[s], op2_reg[s], op3_reg[s]))

    tmp=sl.check()
```

```
        if tmp==sat:
            print ("sat!")
            m=sl.model()
            #print (m)
            print_model(m, R, STEPS, op, op1_reg, op2_reg, op3_reg)
            selftest(m, STEPS, op, op1_reg, op2_reg, op3_reg)
            return True
        else:
            print (tmp)
        return False

    for s in range(len(INPUTS)+len(OUTPUTS), MAX_STEPS):
        if attempt(s):
            return

for i in range(16):
    print ("getting circuit for TT=", format(i & 2**BITS-1, '0'+str(BITS)+'b'))
    find_NAND_only_for_TT ([i])
```

My solution for NAND:

```
3 instructions for OUTPUTS TT (Knuth's representation)= 0000
r0=input                 1100
r1=input                 1010
r2=NAND r1, r1           0101
r3=NAND r1, r2           1111
r4=NAND r3, r3           0000

4 instructions for OUTPUTS TT (Knuth's representation)= 1000
r0=input                 1100
r1=input                 1010
r2=NAND r0, r0           0011
r3=NAND r1, r1           0101
r4=NAND r2, r3           1110
r5=NAND r4, r4           0001

3 instructions for OUTPUTS TT (Knuth's representation)= 0100
r0=input                 1100
r1=input                 1010
r2=NAND r1, r0           0111
r3=NAND r1, r2           1101
r4=NAND r3, r3           0010

1 instructions for OUTPUTS TT (Knuth's representation)= 1100
r0=input                 1100
r1=input                 1010
r2=NAND r0, r0           0011

3 instructions for OUTPUTS TT (Knuth's representation)= 0010
r0=input                 1100
r1=input                 1010
r2=NAND r1, r1           0101
r3=NAND r0, r2           1011
r4=NAND r3, r3           0100

1 instructions for OUTPUTS TT (Knuth's representation)= 1010
r0=input                 1100
r1=input                 1010
r2=NAND r1, r1           0101

4 instructions for OUTPUTS TT (Knuth's representation)= 0110
r0=input                 1100
```

```
r1=input                 1010
r2=NAND r0, r1           0111
r3=NAND r2, r1           1101
r4=NAND r2, r0           1011
r5=NAND r3, r4           0110

1 instructions for OUTPUTS TT (Knuth's representation)= 1110
r0=input                 1100
r1=input                 1010
r2=NAND r0, r1           0111

2 instructions for OUTPUTS TT (Knuth's representation)= 0001
r0=input                 1100
r1=input                 1010
r2=NAND r0, r1           0111
r3=NAND r2, r2           1000

5 instructions for OUTPUTS TT (Knuth's representation)= 1001
r0=input                 1100
r1=input                 1010
r2=NAND r1, r1           0101
r3=NAND r2, r0           1011
r4=NAND r3, r0           0111
r5=NAND r2, r3           1110
r6=NAND r5, r4           1001

2 instructions for OUTPUTS TT (Knuth's representation)= 0101
r0=input                 1100
r1=input                 1010
r2=NAND r1, r1           0101
r3=NAND r2, r2           1010

2 instructions for OUTPUTS TT (Knuth's representation)= 1101
r0=input                 1100
r1=input                 1010
r2=NAND r1, r1           0101
r3=NAND r2, r0           1011

2 instructions for OUTPUTS TT (Knuth's representation)= 0011
r0=input                 1100
r1=input                 1010
r2=NAND r0, r0           0011
r3=NAND r2, r2           1100

2 instructions for OUTPUTS TT (Knuth's representation)= 1011
r0=input                 1100
r1=input                 1010
r2=NAND r0, r1           0111
r3=NAND r1, r2           1101

3 instructions for OUTPUTS TT (Knuth's representation)= 0111
r0=input                 1100
r1=input                 1010
r2=NAND r0, r0           0011
r3=NAND r1, r1           0101
r4=NAND r3, r2           1110

2 instructions for OUTPUTS TT (Knuth's representation)= 1111
r0=input                 1100
r1=input                 1010
r2=NAND r1, r1           0101
```

```
r3=NAND r2, r1           1111
```

My solution for NAND with 0/1 constants:

```
1 instructions for OUTPUTS TT (Knuth's representation)= 0000
r0=input                 1100
r1=input                 1010
r2=input                 0000
r3=input                 1111
r4=NAND r3, r3           0000

4 instructions for OUTPUTS TT (Knuth's representation)= 1000
r0=input                 1100
r1=input                 1010
r2=input                 0000
r3=input                 1111
r4=NAND r1, r1           0101
r5=NAND r0, r0           0011
r6=NAND r4, r5           1110
r7=NAND r6, r6           0001

3 instructions for OUTPUTS TT (Knuth's representation)= 0100
r0=input                 1100
r1=input                 1010
r2=input                 0000
r3=input                 1111
r4=NAND r0, r0           0011
r5=NAND r4, r1           1101
r6=NAND r5, r3           0010

1 instructions for OUTPUTS TT (Knuth's representation)= 1100
r0=input                 1100
r1=input                 1010
r2=input                 0000
r3=input                 1111
r4=NAND r0, r3           0011

3 instructions for OUTPUTS TT (Knuth's representation)= 0010
r0=input                 1100
r1=input                 1010
r2=input                 0000
r3=input                 1111
r4=NAND r0, r1           0111
r5=NAND r4, r0           1011
r6=NAND r5, r3           0100

1 instructions for OUTPUTS TT (Knuth's representation)= 1010
r0=input                 1100
r1=input                 1010
r2=input                 0000
r3=input                 1111
r4=NAND r1, r3           0101

4 instructions for OUTPUTS TT (Knuth's representation)= 0110
r0=input                 1100
r1=input                 1010
r2=input                 0000
r3=input                 1111
r4=NAND r1, r0           0111
r5=NAND r4, r0           1011
r6=NAND r1, r4           1101
r7=NAND r6, r5           0110
```

```
1 instructions for OUTPUTS TT (Knuth's representation)= 1110
r0=input                  1100
r1=input                  1010
r2=input                  0000
r3=input                  1111
r4=NAND r1, r0            0111

2 instructions for OUTPUTS TT (Knuth's representation)= 0001
r0=input                  1100
r1=input                  1010
r2=input                  0000
r3=input                  1111
r4=NAND r1, r0            0111
r5=NAND r4, r4            1000

5 instructions for OUTPUTS TT (Knuth's representation)= 1001
r0=input                  1100
r1=input                  1010
r2=input                  0000
r3=input                  1111
r4=NAND r1, r1            0101
r5=NAND r4, r0            1011
r6=NAND r5, r4            1110
r7=NAND r0, r1            0111
r8=NAND r7, r6            1001

2 instructions for OUTPUTS TT (Knuth's representation)= 0101
r0=input                  1100
r1=input                  1010
r2=input                  0000
r3=input                  1111
r4=NAND r1, r1            0101
r5=NAND r4, r4            1010

2 instructions for OUTPUTS TT (Knuth's representation)= 1101
r0=input                  1100
r1=input                  1010
r2=input                  0000
r3=input                  1111
r4=NAND r0, r1            0111
r5=NAND r0, r4            1011

2 instructions for OUTPUTS TT (Knuth's representation)= 0011
r0=input                  1100
r1=input                  1010
r2=input                  0000
r3=input                  1111
r4=NAND r0, r0            0011
r5=NAND r4, r4            1100

2 instructions for OUTPUTS TT (Knuth's representation)= 1011
r0=input                  1100
r1=input                  1010
r2=input                  0000
r3=input                  1111
r4=NAND r1, r0            0111
r5=NAND r1, r4            1101

3 instructions for OUTPUTS TT (Knuth's representation)= 0111
r0=input                  1100
```

```
r1=input                1010
r2=input                0000
r3=input                1111
r4=NAND r1, r1          0101
r5=NAND r0, r0          0011
r6=NAND r5, r4          1110

1 instructions for OUTPUTS TT (Knuth's representation)= 1111
r0=input                1100
r1=input                1010
r2=input                0000
r3=input                1111
r4=NAND r2, r2          1111
```

My solution for ANDN:

```
1 instructions for OUTPUTS TT (Knuth's representation)= 0000
r0=input                1100
r1=input                1010
r2=ANDN r1, r1          0000

1 instructions for OUTPUTS TT (Knuth's representation)= 0100
r0=input                1100
r1=input                1010
r2=ANDN r0, r1          0010

1 instructions for OUTPUTS TT (Knuth's representation)= 0010
r0=input                1100
r1=input                1010
r2=ANDN r1, r0          0100

2 instructions for OUTPUTS TT (Knuth's representation)= 0001
r0=input                1100
r1=input                1010
r2=ANDN r0, r1          0010
r3=ANDN r2, r1          1000

2 instructions for OUTPUTS TT (Knuth's representation)= 0101
r0=input                1100
r1=input                1010
r2=ANDN r1, r1          0000
r3=ANDN r2, r1          1010

2 instructions for OUTPUTS TT (Knuth's representation)= 0011
r0=input                1100
r1=input                1010
r2=ANDN r0, r1          0010
r3=ANDN r2, r0          1100
```

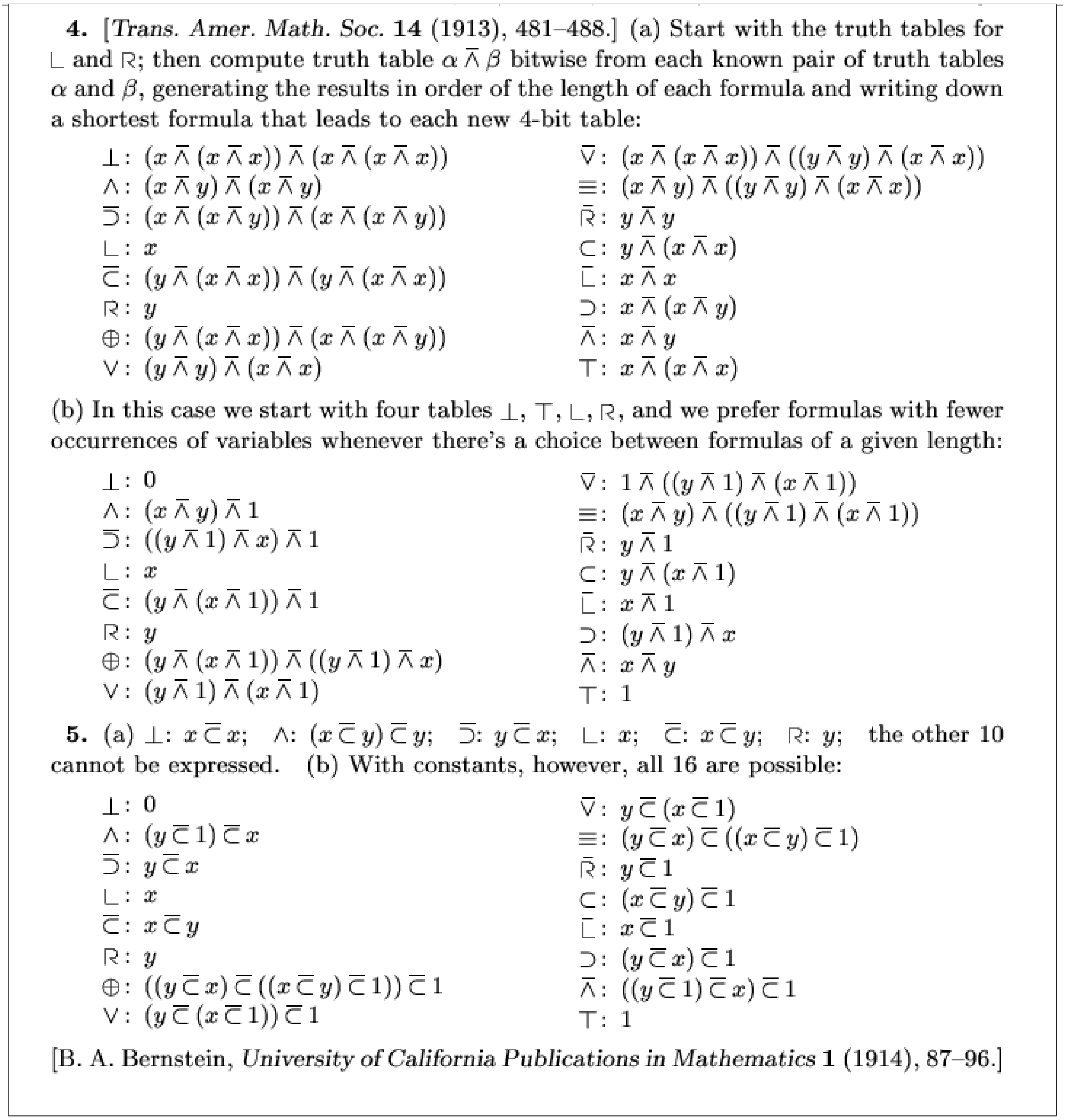
My solution for ANDN with 0/1 constants:

```
1 instructions for OUTPUTS TT (Knuth's representation)= 0000
r0=input                1100
r1=input                1010
r2=input                0000
r3=input                1111
r4=ANDN r3, r2          0000

2 instructions for OUTPUTS TT (Knuth's representation)= 1000
r0=input                1100
r1=input                1010
r2=input                0000
r3=input                1111
```

```
r4=ANDN r0, r3           0011
r5=ANDN r1, r4           0001

1 instructions for OUTPUTS TT (Knuth's representation)= 0100
r0=input                 1100
r1=input                 1010
r2=input                 0000
r3=input                 1111
r4=ANDN r0, r1           0010

1 instructions for OUTPUTS TT (Knuth's representation)= 1100
r0=input                 1100
r1=input                 1010
r2=input                 0000
r3=input                 1111
r4=ANDN r0, r3           0011

1 instructions for OUTPUTS TT (Knuth's representation)= 0010
r0=input                 1100
r1=input                 1010
r2=input                 0000
r3=input                 1111
r4=ANDN r1, r0           0100

1 instructions for OUTPUTS TT (Knuth's representation)= 1010
r0=input                 1100
r1=input                 1010
r2=input                 0000
r3=input                 1111
r4=ANDN r1, r3           0101

5 instructions for OUTPUTS TT (Knuth's representation)= 0110
r0=input                 1100
r1=input                 1010
r2=input                 0000
r3=input                 1111
r4=ANDN r0, r3           0011
r5=ANDN r1, r4           0001
r6=ANDN r4, r1           1000
r7=ANDN r6, r3           0111
r8=ANDN r5, r7           0110

3 instructions for OUTPUTS TT (Knuth's representation)= 1110
r0=input                 1100
r1=input                 1010
r2=input                 0000
r3=input                 1111
r4=ANDN r1, r0           0100
r5=ANDN r4, r0           1000
r6=ANDN r5, r3           0111

2 instructions for OUTPUTS TT (Knuth's representation)= 0001
r0=input                 1100
r1=input                 1010
r2=input                 0000
r3=input                 1111
r4=ANDN r0, r1           0010
r5=ANDN r4, r1           1000

4 instructions for OUTPUTS TT (Knuth's representation)= 1001
r0=input                 1100
```

```
r1=input                  1010
r2=input                  0000
r3=input                  1111
r4=ANDN r1, r0            0100
r5=ANDN r0, r1            0010
r6=ANDN r4, r3            1011
r7=ANDN r5, r6            1001

1 instructions for OUTPUTS TT (Knuth's representation)= 0101
r0=input                  1100
r1=input                  1010
r2=input                  0000
r3=input                  1111
r4=ANDN r2, r1            1010

2 instructions for OUTPUTS TT (Knuth's representation)= 1101
r0=input                  1100
r1=input                  1010
r2=input                  0000
r3=input                  1111
r4=ANDN r1, r0            0100
r5=ANDN r4, r3            1011

1 instructions for OUTPUTS TT (Knuth's representation)= 0011
r0=input                  1100
r1=input                  1010
r2=input                  0000
r3=input                  1111
r4=ANDN r2, r0            1100

2 instructions for OUTPUTS TT (Knuth's representation)= 1011
r0=input                  1100
r1=input                  1010
r2=input                  0000
r3=input                  1111
r4=ANDN r0, r1            0010
r5=ANDN r4, r3            1101

3 instructions for OUTPUTS TT (Knuth's representation)= 0111
r0=input                  1100
r1=input                  1010
r2=input                  0000
r3=input                  1111
r4=ANDN r0, r3            0011
r5=ANDN r1, r4            0001
r6=ANDN r5, r3            1110

1 instructions for OUTPUTS TT (Knuth's representation)= 1111
r0=input                  1100
r1=input                  1010
r2=input                  0000
r3=input                  1111
r4=ANDN r2, r3            1111
```

Correct answers from TAOCP:



4. [*Trans. Amer. Math. Soc.* **14** (1913), 481–488.] (a) Start with the truth tables for $\llcorner$ and R; then compute truth table $\alpha \overline{\wedge} \beta$ bitwise from each known pair of truth tables α and β, generating the results in order of the length of each formula and writing down a shortest formula that leads to each new 4-bit table:

$\perp\colon (x \overline{\wedge} (x \overline{\wedge} x)) \overline{\wedge} (x \overline{\wedge} (x \overline{\wedge} x))$

$\wedge\colon (x \overline{\wedge} y) \overline{\wedge} (x \overline{\wedge} y)$

$\overline{\supset}\colon (x \overline{\wedge} (x \overline{\wedge} y)) \overline{\wedge} (x \overline{\wedge} (x \overline{\wedge} y))$

$\llcorner\colon x$

$\overline{\subset}\colon (y \overline{\wedge} (x \overline{\wedge} x)) \overline{\wedge} (y \overline{\wedge} (x \overline{\wedge} x))$

$\mathsf{R}\colon y$

$\oplus\colon (y \overline{\wedge} (x \overline{\wedge} x)) \overline{\wedge} (x \overline{\wedge} (x \overline{\wedge} y))$

$\vee\colon (y \overline{\wedge} y) \overline{\wedge} (x \overline{\wedge} x)$

$\overline{\vee}\colon (x \overline{\wedge} (x \overline{\wedge} x)) \overline{\wedge} ((y \overline{\wedge} y) \overline{\wedge} (x \overline{\wedge} x))$

$\equiv\colon (x \overline{\wedge} y) \overline{\wedge} ((y \overline{\wedge} y) \overline{\wedge} (x \overline{\wedge} x))$

$\overline{\mathsf{R}}\colon y \overline{\wedge} y$

$\subset\colon y \overline{\wedge} (x \overline{\wedge} x)$

$\overline{\llcorner}\colon x \overline{\wedge} x$

$\supset\colon x \overline{\wedge} (x \overline{\wedge} y)$

$\overline{\wedge}\colon x \overline{\wedge} y$

$\top\colon x \overline{\wedge} (x \overline{\wedge} x)$

(b) In this case we start with four tables $\perp$, $\top$, $\llcorner$, R, and we prefer formulas with fewer occurrences of variables whenever there's a choice between formulas of a given length:

$\perp\colon 0$

$\wedge\colon (x \overline{\wedge} y) \overline{\wedge} 1$

$\overline{\supset}\colon ((y \overline{\wedge} 1) \overline{\wedge} x) \overline{\wedge} 1$

$\llcorner\colon x$

$\overline{\subset}\colon (y \overline{\wedge} (x \overline{\wedge} 1)) \overline{\wedge} 1$

$\mathsf{R}\colon y$

$\oplus\colon (y \overline{\wedge} (x \overline{\wedge} 1)) \overline{\wedge} ((y \overline{\wedge} 1) \overline{\wedge} x)$

$\vee\colon (y \overline{\wedge} 1) \overline{\wedge} (x \overline{\wedge} 1)$

$\overline{\vee}\colon 1 \overline{\wedge} ((y \overline{\wedge} 1) \overline{\wedge} (x \overline{\wedge} 1))$

$\equiv\colon (x \overline{\wedge} y) \overline{\wedge} ((y \overline{\wedge} 1) \overline{\wedge} (x \overline{\wedge} 1))$

$\overline{\mathsf{R}}\colon y \overline{\wedge} 1$

$\subset\colon y \overline{\wedge} (x \overline{\wedge} 1)$

$\overline{\llcorner}\colon x \overline{\wedge} 1$

$\supset\colon (y \overline{\wedge} 1) \overline{\wedge} x$

$\overline{\wedge}\colon x \overline{\wedge} y$

$\top\colon 1$

5. (a) $\perp\colon x \overline{\subset} x$; $\wedge\colon (x \overline{\subset} y) \overline{\subset} y$; $\overline{\supset}\colon y \overline{\subset} x$; $\llcorner\colon x$; $\overline{\subset}\colon x \overline{\subset} y$; $\mathsf{R}\colon y$; the other 10 cannot be expressed. (b) With constants, however, all 16 are possible:

$\perp\colon 0$

$\wedge\colon (y \overline{\subset} 1) \overline{\subset} x$

$\overline{\supset}\colon y \overline{\subset} x$

$\llcorner\colon x$

$\overline{\subset}\colon x \overline{\subset} y$

$\mathsf{R}\colon y$

$\oplus\colon ((y \overline{\subset} x) \overline{\subset} ((x \overline{\subset} y) \overline{\subset} 1)) \overline{\subset} 1$

$\vee\colon (y \overline{\subset} (x \overline{\subset} 1)) \overline{\subset} 1$

$\overline{\vee}\colon y \overline{\subset} (x \overline{\subset} 1)$

$\equiv\colon (y \overline{\subset} x) \overline{\subset} ((x \overline{\subset} y) \overline{\subset} 1)$

$\overline{\mathsf{R}}\colon y \overline{\subset} 1$

$\subset\colon (x \overline{\subset} y) \overline{\subset} 1$

$\overline{\llcorner}\colon x \overline{\subset} 1$

$\supset\colon (y \overline{\subset} x) \overline{\subset} 1$

$\overline{\wedge}\colon ((y \overline{\subset} 1) \overline{\subset} x) \overline{\subset} 1$

$\top\colon 1$

[B. A. Bernstein, *University of California Publications in Mathematics* **1** (1914), 87–96.]

Figure 15.4: Page 51

My solutions are slightly different: I haven't "pass through" instruction, so sometimes a value is copied from the input to the output using NAND/ANDN. Also, my versions are sometimes different, but correct and has the same length.

15.2 Program synthesis

Program synthesis is a process of automatic program generation, in accordance with some specific goals.

15.2.1 Synthesis of simple program using Z3 SMT-solver

Sometimes, multiplication operation can be replaced with a several operations of shifting/addition/subtraction. Compilers do so, because pack of instructions can be executed faster.

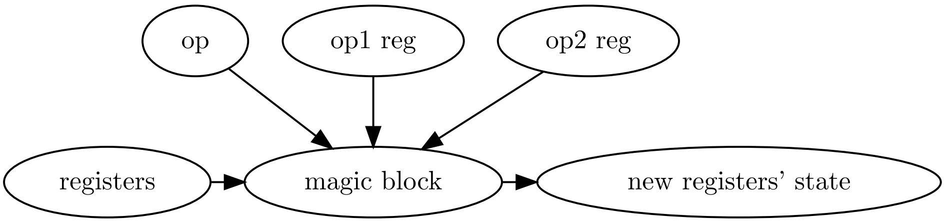
For example, multiplication by 19 is replaced by GCC 5.4 with a pair of instructions: `lea edx, [eax+eax*8]` and `lea eax, [eax+edx*2]`. This is sometimes also called "superoptimization".

Let's see if we can find a shortest possible instructions pack for some specified multiplier.

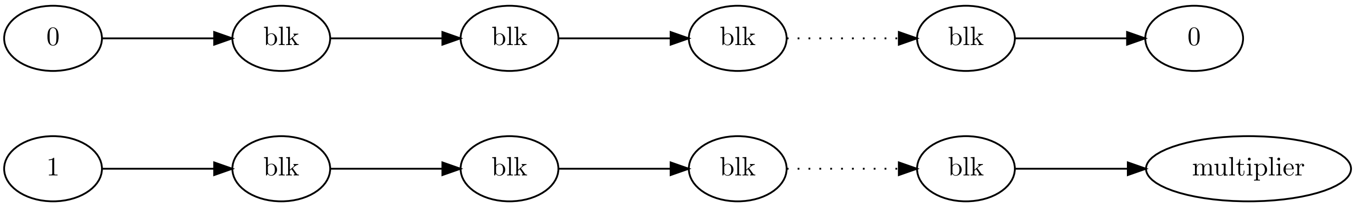
As I've already wrote once, SMT-solver can be seen as a solver of huge systems of equations. The task is to construct such system of equations, which, when solved, could produce a short program. I will use electronics analogy here, it can make things a little simpler.

First of all, what our program will be consisting of? There will be 3 operations allowed: ADD/SUB/SHL. Only registers allowed as operands, except for the second operand of SHL (which could be in 1..31 range). Each register will be assigned only once (as in SSA).

And there will be some "magic block", which takes all previous register states, it also takes operation type, operands and produces a value of next register's state.



Now let's take a look on our schematics on top level:

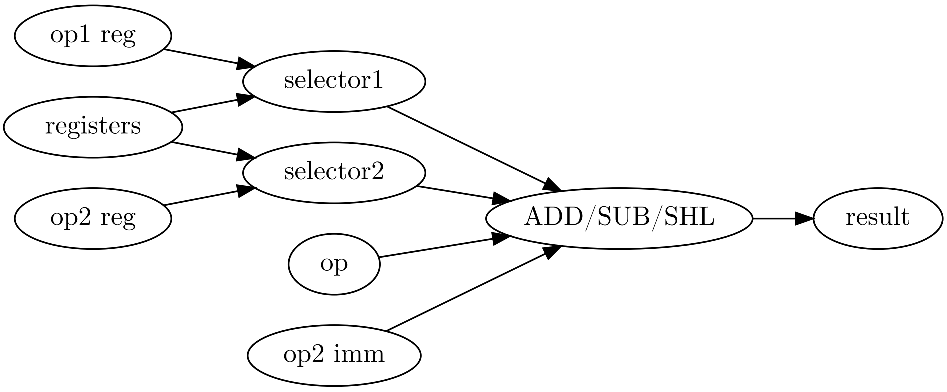


Each block takes previous state of registers and produces new states. There are two chains. First chain takes 0 as state of R0 at the very beginning, and the chain is supposed to produce 0 at the end (since zero multiplied by any value is still zero). The second chain takes 1 and must produce multiplier as the state of very last register (since 1 multiplied by multiplier must equal to multiplier).

Each block is "controlled" by operation type, operands, etc. For each column, there is each own set.

Now you can view these two chains as two equations. The ultimate goal is to find such state of all operation types and operands, so the first chain will equal to 0, and the second to multiplier.

Let's also take a look into "magic block" inside:



It's like ALU.

Each selector can be viewed as a simple multipositional switch. If operation is SHL, a value in range of 1..31 is used as second operand.

So you can imagine this electric circuit and your goal is to turn all switches in such a state, so two chains will have 0 and multiplier on output. This sounds like logic puzzle in some way. Now we will try to use Z3 to solve this puzzle.

First, we define all variables:

```
R=[[BitVec('S_s%d_c%d' % (s, c), 32) for s in range(MAX_STEPS)] for c in range (
    CHAINS)]
```

```
op=[Int('op_s%d' % s) for s in range(MAX_STEPS)]
op1_reg=[Int('op1_reg_s%d' % s) for s in range(MAX_STEPS)]
op2_reg=[Int('op2_reg_s%d' % s) for s in range(MAX_STEPS)]
op2_imm=[BitVec('op2_imm_s%d' % s, 32) for s in range(MAX_STEPS)]
```

R[][] is registers state for each chain and each step.
On contrary, op/op1_reg/op2_reg/op2_imm variables are defined for each step, but for both chains, since both chains at each column has the same operation/operands.

Now we must limit count of operations, and also, register's number for each step must not be bigger than step number, in other words, instruction at each step is allowed to access only registers which were already set before:

```
for s in range(1, STEPS):
    # for each step
    sl.add(And(op[s]>=0, op[s]<=2))
    sl.add(And(op1_reg[s]>=0, op1_reg[s]<s))
    sl.add(And(op2_reg[s]>=0, op2_reg[s]<s))
    sl.add(And(op2_imm[s]>=1, op2_imm[s]<=31))
```

Fix register of first step for both chains:

```
for c in range(CHAINS):
    # for each chain:
    sl.add(R[c][0]==chain_inputs[c])
    sl.add(R[c][STEPS-1]==chain_inputs[c]*multiplier)
```

Now let's add "magic blocks":

```
for s in range(1, STEPS):
    sl.add(R[c][s]==simulate_op(R,c, op[s], op1_reg[s], op2_reg[s], op2_imm[s]))
```

Now how "magic block" is defined?

```
def selector(R, c, s):
    # for all MAX_STEPS:
    return If(s==0, R[c][0],
             If(s==1, R[c][1],
             If(s==2, R[c][2],
             If(s==3, R[c][3],
             If(s==4, R[c][4],
             If(s==5, R[c][5],
             If(s==6, R[c][6],
             If(s==7, R[c][7],
             If(s==8, R[c][8],
             If(s==9, R[c][9],
                 0)))))))))) # default

def simulate_op(R, c, op, op1_reg, op2_reg, op2_imm):
    op1_val=selector(R,c,op1_reg)
    return If(op==0, op1_val + selector(R, c, op2_reg),
           If(op==1, op1_val - selector(R, c, op2_reg),
           If(op==2, op1_val << op2_imm,
               0))) # default
```

This is very important to understand: if the operation is ADD/SUB, op2_imm's value is just ignored. Otherwise, if operation is SHL, value of op2_reg is ignored. Just like in case of digital circuit.

The code: https://smt.st/current_tree/synth/pgm/mult/mult.py

Now let's see how it works:

```
% ./mult.py 12
multiplier= 12
attempt, STEPS= 2
unsat
attempt, STEPS= 3
unsat
```

```
attempt , STEPS= 4
sat !
r1=SHL r0 , 2
r2=SHL r1 , 1
r3=ADD r1 , r2
tests are OK
```

The first step is always a step containing 0/1, or, r0. So when our solver reporting about 4 steps, this means 3 instructions.

Something harder:

```
% ./mult.py 123
multiplier= 123
attempt , STEPS= 2
unsat
attempt , STEPS= 3
unsat
attempt , STEPS= 4
unsat
attempt , STEPS= 5
sat !
r1=SHL r0 , 2
r2=SHL r1 , 5
r3=SUB r2 , r1
r4=SUB r3 , r0
tests are OK
```

Now the code multiplying by 1234:

```
r1=SHL r0 , 6
r2=ADD r0 , r1
r3=ADD r2 , r1
r4=SHL r2 , 4
r5=ADD r2 , r3
r6=ADD r5 , r4
```

Looks great, but it took ≈ 23 seconds to find it on my Intel Xeon CPU E3-1220 @ 3.10GHz. I agree, this is far from practical usage. Also, I'm not quite sure that this piece of code will work faster than a single multiplication instruction. But anyway, it's a good demonstration of SMT solvers capabilities.

The code multiplying by 12345 (≈ 150 seconds):

```
r1=SHL r0 , 5
r2=SHL r0 , 3
r3=SUB r2 , r1
r4=SUB r1 , r3
r5=SHL r3 , 9
r6=SUB r4 , r5
r7=ADD r0 , r6
```

Multiplication by 123456 (≈ 8 minutes!):

```
r1=SHL r0 , 9
r2=SHL r0 , 13
r3=SHL r0 , 2
r4=SUB r1 , r2
r5=SUB r3 , r4
r6=SHL r5 , 4
r7=ADD r1 , r6
```

What multipliers are possible? I tried all multipliers in the [1..2530] range with success. Further work: try more.

Few notes

I've removed SHR instruction support, simply because the code multiplying by a constant makes no use of it. Even more: it's not a problem to add support of constants as second operand for all instructions, but again, you wouldn't find a piece of code which does this job and uses some additional constants. Or maybe I wrong?

Of course, for another job you'll need to add support of constants and other operations. But at the same time, it will work slower and slower. So I had to keep ISA[1] of this toy CPU as compact as possible.

The code

https://smt.st/current_tree/synth/pgm/mult

See also

Multiplication using a series of additions also called *addition chain*. [See *Alexander A. Stepanov, Daniel E. Rose – From Mathematics to Generic Programming*, section *2.1 Egyptian Multiplication*.]

15.2.2 Rockey dongle: finding unknown algorithm using only input/output pairs

Some smartcards can execute Java or .NET code - that's the way to hide your sensitive algorithm into chip that is very hard to break (decapsulate). For example, one may encrypt/decrypt data files by hidden crypto algorithm rendering software piracy of such software close to impossible—an encrypted date file created on software with connected smartcard would be impossible to decrypt on cracked version of the same software. (This leads to many nuisances, though.)

That's what is called *black box*.

Some software protection dongles offers this functionality too. One example is Rockey 4[2].

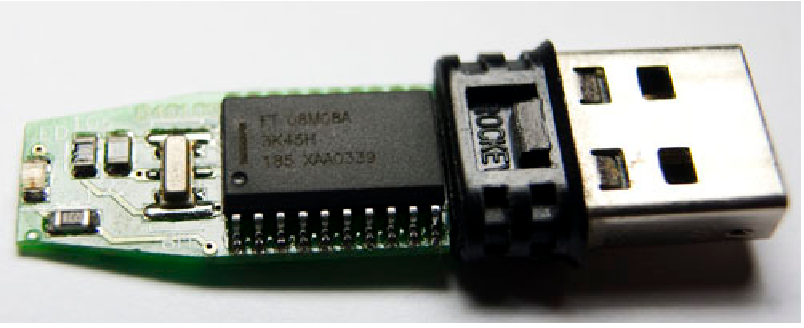


Figure 15.5: Rockey 4 dongle

This is a small dongle connected via USB. Is contain some user-defined memory but also memory for user algorithms.

The virtual (toy) CPU for these algorithms is very simple: it offer only 8 16-bit registers (however, only 4 can be set and read) and 8 operations (addition, subtraction, cyclic left shifting, multiplication, OR, XOR, AND, negation).

Second instruction argument can be a constant (from 0 to 63) instead of register.

Each algorithm is described by string like
`A=A+B, B=C*13, D=D^A, C=B*55, C=C&A, D=D|A, A=A*9, A=A&B.`

There are no memory, stack, conditional/unconditional jumps, etc.

Each algorithm, obviously, can't have side effects, so they are actually *pure functions* and their results can be *memoized*.

By the way, as it has been mentioned in Rockey 4 manual, first and last instruction cannot have constants. Maybe that's because these fields used for some internal data: each algorithm start and end should be marked somehow internally anyway.

Would it be possible to reveal hidden impossible-to-read algorithm only by recording input/output dongle traffic? Common sense tell us "no". But we can try anyway.

Since, my goal wasn't to break into some Rockey-protected software, I was interesting only in limits (which algorithms could we find), so I make some things simpler: we will work with only 4 16-bit registers, and there will be only 6 operations (add, subtract, multiply, OR, XOR, AND).

Let's first calculate, how much information will be used in brute-force case.

There are 384 of all possible instructions in `reg=reg,op,reg` format for 4 registers and 6 operations, and also 6144 instructions in `reg=reg,op,constant` format. Remember that constant limited to 63 as maximal value? That help us for a little.

So, there are 6528 of all possible instructions. This mean, there are $\approx 1.1 \cdot 10^{19}$ 5-instruction algorithms. That's too much.

[1] Instruction Set Architecture

[2] http://www.rockey.nl/en/rockey.html

How can we express each instruction as system of equations? While remembering some school mathematics, I wrote this:

```
Function one\_step()=

# Each Bx is integer, but may be only 0 or 1.

# only one of B1..B4 and B5..B9 can be set
reg1=B1*A + B2*B + B3*C + B4*D
reg_or_constant2=B5*A + B6*B + B7*C + B8*D + B9*constant
reg1 should not be equal to reg_or_constant2

# Only one of B10..B15 can be set
result=result+B10*(reg1*reg2)
result=result+B11*(reg1^reg2)
result=result+B12*(reg1+reg2)
result=result+B13*(reg1-reg2)
result=result+B14*(reg1|reg2)
result=result+B15*(reg1&reg2)

B16 - true if register isn't updated in this part
B17 - true if register is updated in this part
(B16 cannot be equal to B17)
A=B16*A + B17*result
B=B18*A + B19*result
C=B20*A + B21*result
D=B22*A + B23*result
```

That's how we can express each instruction in algorithm.

5-instructions algorithm can be expressed like this:
`one_step (one_step (one_step (one_step (one_step (input_registers))))).`

Let's also add five known input/output pairs and we'll get system of equations like this:

```
one_step (one_step (one_step (one_step (one_step (input_1)))))==output_1
one_step (one_step (one_step (one_step (one_step (input_2)))))==output_2
one_step (one_step (one_step (one_step (one_step (input_3)))))==output_3
one_step (one_step (one_step (one_step (one_step (input_4)))))==output_4
.. etc
```

So the question now is to find $5 \cdot 23$ Boolean values satisfying known input/output pairs.

I wrote small utility to probe Rockey 4 algorithm with random numbers, it produce results in form:

```
RY_CALCULATE1: (input) p1=30760 p2=18484 p3=41200 p4=61741 (output) p1=49244 p2=11312
     p3=27587 p4=12657
RY_CALCULATE1: (input) p1=51139 p2=7852 p3=53038 p4=49378 (output) p1=58991 p2=34134
    p3=40662 p4=9869
RY_CALCULATE1: (input) p1=60086 p2=52001 p3=13352 p4=45313 (output) p1=46551 p2=42504
     p3=61472 p4=1238
RY_CALCULATE1: (input) p1=48318 p2=6531 p3=51997 p4=30907 (output) p1=54849 p2=20601
    p3=31271 p4=44794
```

p1/p2/p3/p4 are just another names for A/B/C/D registers.

Now let's start with Z3. We will need to express Rockey 4 toy CPU in Z3Py (Z3 Python API) terms.

It can be said, my Python script is divided into two parts:

- constraint definitions (like, *output_1 should be n for input_1=m*, *constant cannot be greater than 63*, etc);
- functions constructing system of equations.

This piece of code define some kind of *structure* consisting of 4 named 16-bit variables, each represent register in our toy CPU.

```
Registers_State=Datatype ('Registers_State')
Registers_State.declare('cons', ('A', BitVecSort(16)), ('B', BitVecSort(16)), ('C',
    BitVecSort(16)), ('D', BitVecSort(16)))
```

```
Registers_State=Registers_State.create()
```

These enumerations define two new types (or *sorts* in Z3's terminology):

```
Operation, (OP_MULT, OP_MINUS, OP_PLUS, OP_XOR, OP_OR, OP_AND) = EnumSort('Operation
    ', ('OP_MULT', 'OP_MINUS', 'OP_PLUS', 'OP_XOR', 'OP_OR', 'OP_AND'))

Register, (A, B, C, D) = EnumSort('Register', ('A', 'B', 'C', 'D'))
```

This part is very important, it defines all variables in our system of equations. `op_step` is type of operation in instruction. `reg_or_constant` is selector between register and constant in second argument — *False* if it's a register and *True* if it's a constant. `reg_step` is a destination register of this instruction. `reg1_step` and `reg2_step` are just registers at arg1 and arg2. `constant_step` is constant (in case it's used in instruction instead of arg2).

```
op_step=[Const('op_step%s' % i, Operation) for i in range(STEPS)]
reg_or_constant_step=[Bool('reg_or_constant_step%s' % i) for i in range(STEPS)]
reg_step=[Const('reg_step%s' % i, Register) for i in range(STEPS)]
reg1_step=[Const('reg1_step%s' % i, Register) for i in range(STEPS)]
reg2_step=[Const('reg2_step%s' % i, Register) for i in range(STEPS)]
constant_step = [BitVec('constant_step%s' % i, 16) for i in range(STEPS)]
```

Adding constraints is very simple. Remember, I wrote that each constant cannot be larger than 63?

```
# according to Rockey 4 dongle manual, arg2 in first and last instructions cannot be
    a constant
s.add (reg_or_constant_step[0]==False)
s.add (reg_or_constant_step[STEPS-1]==False)

...

for x in range(STEPS):
   s.add (constant_step[x]>=0, constant_step[x]<=63)
```

Known input/output values are added as constraints too.

So far, I used the fact that multiplication by zero results to zero. As Leopold Kronecker said: "God made the integers; all else is the work of man." This works in place of *If*'s, but clumsy.

We can just use "If" operation instead.

Now let's see how to construct our system of equations:

```
# Register, Registers_State -> int
def register_selector (register, input_registers):
    return If(register==A, Registers_State.A(input_registers),
           If(register==B, Registers_State.B(input_registers),
           If(register==C, Registers_State.C(input_registers),
           If(register==D, Registers_State.D(input_registers),
                           0)))) # default
```

This function returning corresponding register value from *structure*. Needless to say, the code above is not executed. `If()` is Z3Py function. The code only declares the function, which will be used in another. Expression declaration resembling LISP PL in some way.

Here is another function where `register_selector()` is used:

```
# Bool, Register, Registers_State, int -> int
def register_or_constant_selector (register_or_constant, register, input_registers,
    constant):
    return If(register_or_constant==False, register_selector(register,
        input_registers), constant)
```

The code here is never executed too. It only constructs one small piece of very big expression. But for the sake of simplicity, one can think all these functions will be called during bruteforce search, many times, at fastest possible speed.

```
# Operation, Bool, Register, Register, Int, Registers_State -> int
def one_op (op, register_or_constant, reg1, reg2, constant, input_registers):
    arg1=register_selector(reg1, input_registers)
    arg2=register_or_constant_selector (register_or_constant , reg2 , input_registers ,
        constant)
    return If(op==OP_MULT ,   arg1*arg2 ,
           If(op==OP_MINUS ,  arg1-arg2 ,
           If(op==OP_PLUS ,   arg1+arg2 ,
           If(op==OP_XOR ,    arg1^arg2 ,
           If(op==OP_OR ,     arg1|arg2 ,
           If(op==OP_AND ,    arg1&arg2 ,
                         0)))))) # default
```

Here is the expression describing each instruction. `new_val` will be assigned to destination register, while all other registers' values are copied from input registers' state:

```
# Bool , Register , Operation , Register , Register , Int , Registers_State ->
    Registers_State
def one_step (register_or_constant , register_assigned_in_this_step , op , reg1 , reg2 ,
    constant , input_registers):
    new_val=one_op(op , register_or_constant , reg1 , reg2 , constant , input_registers)
    return If (register_assigned_in_this_step==A , Registers_State.cons (new_val ,
                                Registers_State.B(input_registers) ,
                                Registers_State.C(input_registers) ,
                                Registers_State.D(input_registers)) ,
           If (register_assigned_in_this_step==B , Registers_State.cons (
               Registers_State.A(input_registers) ,
                                new_val ,
                                Registers_State.C(input_registers) ,
                                Registers_State.D(input_registers)) ,
           If (register_assigned_in_this_step==C , Registers_State.cons (
               Registers_State.A(input_registers) ,
                                Registers_State.B(input_registers) ,
                                new_val ,
                                Registers_State.D(input_registers)) ,
           If (register_assigned_in_this_step==D , Registers_State.cons (
               Registers_State.A(input_registers) ,
                                Registers_State.B(input_registers) ,
                                Registers_State.C(input_registers) ,
                                new_val) ,
                                               Registers_State.cons(0 ,0 ,0 ,0))))) #
                                                   default
```

This is the last function describing a whole n-step program:

```
def program(input_registers , STEPS):
    cur_input=input_registers
    for x in range(STEPS):
        cur_input=one_step (reg_or_constant_step[x] , reg_step[x] , op_step[x] ,
            reg1_step[x] , reg2_step[x] , constant_step[x] , cur_input)
    return cur_input
```

Again, for the sake of simplicity, it can be said, now Z3 will try each possible registers/operations/constants against this expression to find such combination which satisfy all input/output pairs. Sounds absurd, but this is close to reality. SAT/SMT-solvers indeed tries them all. But the trick is to prune search tree as early as possible, so it will work for some reasonable time. And this is hardest problem for solvers.

Now let's start with very simple 3-step algorithm: `B=A^D, C=D*D, D=A*C`. Please note: register `A` left unchanged. I programmed Rockey 4 dongle with the algorithm, and recorded algorithm outputs are:

```
RY_CALCULATE1: (input) p1=8803 p2=59946 p3=36002 p4=44743 (output) p1=8803 p2=36004
    p3=7857 p4=24691
RY_CALCULATE1: (input) p1=5814 p2=55512 p3=52155 p4=55813 (output) p1=5814 p2=52403
    p3=33817 p4=4038
RY_CALCULATE1: (input) p1=25206 p2=2097 p3=55906 p4=22705 (output) p1=25206 p2=15047
    p3=10849 p4=43702
RY_CALCULATE1: (input) p1=10044 p2=14647 p3=27923 p4=7325 (output) p1=10044 p2=15265
    p3=47177 p4=20508
```

```
RY_CALCULATE1: (input) p1=15267 p2=2690 p3=47355 p4=56073 (output) p1=15267 p2=57514
    p3=26193 p4=53395
```

It took about one second and only 5 pairs above to find algorithm (on my quad-core Xeon E3-1220 3.1GHz, however, Z3 solver working in single-thread mode):

```
B = A ^ D
C = D * D
D = C * A
```

Note the last instruction: `C` and `A` registers are swapped comparing to version I wrote by hand. But of course, this instruction is working in the same way, because multiplication is commutative operation.

Now if I try to find 4-step program satisfying to these values, my script will offer this:

```
B = A ^ D
C = D * D
D = A * C
A = A | A
```

...and that's really fun, because the last instruction do nothing with value in register `A`, it's like NOP[3]—but still, algorithm is correct for all values given.

Here is another 5-step algorithm: `B=B^D, C=A*22, A=B*19, A=A&42, D=B&C` and values:

```
RY_CALCULATE1: (input) p1=61876 p2=28737 p3=28636 p4=50362 (output) p1=32 p2=46331 p3
    =50552 p4=33912
RY_CALCULATE1: (input) p1=46843 p2=43355 p3=39078 p4=24552 (output) p1=8 p2=63155 p3
    =47506 p4=45202
RY_CALCULATE1: (input) p1=22425 p2=51432 p3=40836 p4=14260 (output) p1=0 p2=65372 p3
    =34598 p4=34564
RY_CALCULATE1: (input) p1=44214 p2=45766 p3=19778 p4=59924 (output) p1=2 p2=22738 p3
    =55204 p4=20608
RY_CALCULATE1: (input) p1=27348 p2=49060 p3=31736 p4=59576 (output) p1=0 p2=22300 p3
    =11832 p4=1560
```

It took 37 seconds and we've got:

```
B = D ^ B
C = A * 22
A = B * 19
A = A & 42
D = C & B
```

`A=A&42` was correctly deduced (look at these five p1's at output (assigned to output `A` register): 32,8,0,2,0)

6-step algorithm `A=A+B, B=C*13, D=D^A, C=C&A, D=D|B, A=A&B` and values:

```
RY_CALCULATE1: (input) p1=4110 p2=35411 p3=54308 p4=47077 (output) p1=32832 p2=50644
    p3=36896 p4=60884
RY_CALCULATE1: (input) p1=12038 p2=7312 p3=39626 p4=47017 (output) p1=18434 p2=56386
    p3=2690 p4=64639
RY_CALCULATE1: (input) p1=48763 p2=27663 p3=12485 p4=20563 (output) p1=10752 p2=31233
     p3=8320 p4=31449
RY_CALCULATE1: (input) p1=33174 p2=38937 p3=54005 p4=38871 (output) p1=4129 p2=46705
    p3=4261 p4=48761
RY_CALCULATE1: (input) p1=46587 p2=36275 p3=6090 p4=63976 (output) p1=258 p2=13634 p3
    =906 p4=48966
```

90 seconds and we've got:

```
A = A + B
B = C * 13
D = D ^ A
D = B | D
C = C & A
A = B & A
```

[3] No Operation

But that was simple, however. Some 6-step algorithms are not possible to find, for example: `A=A^B, A=A*9, A=A^C, A=A*19, A=A^D, A=A&B`. Solver was working too long (up to several hours), so I didn't even know is it possible to find it anyway.

Conclusion

This is in fact an exercise in program synthesis.

Some short algorithms for tiny CPUs are really possible to find using so small set set of data. Of course it's still not possible to reveal some complex algorithm, but this method definitely should not be ignored.

The files

Rockey 4 dongle programmer and reader, Rockey 4 manual, Z3Py script for finding algorithms, input/output pairs: https://smt.st/current_tree/synth/pgm/rockey

Further work

Perhaps, constructing LISP-like S-expression can be better than a program for toy-level CPU.

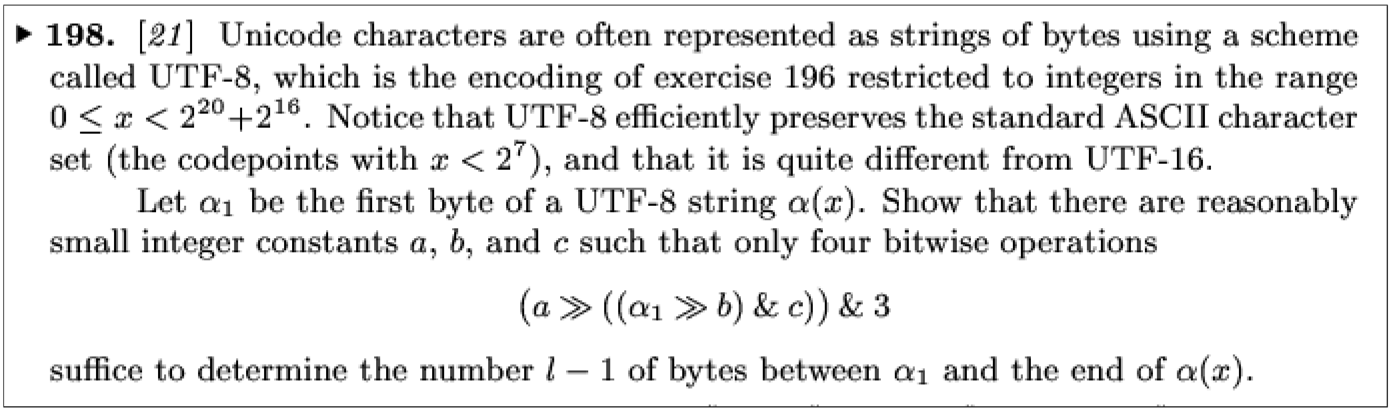
It's also possible to start with smaller constants and then proceed to bigger. This is somewhat similar to increasing password length in password brute-force cracking.

Exercise

https://challenges.re/25/.

15.2.3 TAOCP 7.1.3 Exercise 198, UTF-8 encoding and program synthesis by sketching

Found this exercise in TAOCP 7.1.3 (Bitwise Tricks and Techniques):



▸ **198.** [*21*] Unicode characters are often represented as strings of bytes using a scheme called UTF-8, which is the encoding of exercise 196 restricted to integers in the range $0 \leq x < 2^{20}+2^{16}$. Notice that UTF-8 efficiently preserves the standard ASCII character set (the codepoints with $x < 2^7$), and that it is quite different from UTF-16.

Let α_1 be the first byte of a UTF-8 string $\alpha(x)$. Show that there are reasonably small integer constants a, b, and c such that only four bitwise operations

$$(a \gg ((\alpha_1 \gg b) \,\&\, c)) \,\&\, 3$$

suffice to determine the number $l-1$ of bytes between α_1 and the end of $\alpha(x)$.

Figure 15.6: Exercise from TAOCP book

This is like program synthesis by sketching: you give a sketch with several "holes" missing and ask some automated software to fill the "holes". In our case, a, b and c are "holes".

Let's find them using Z3:

```
#!/usr/bin/env python3
from z3 import *

a, b, c=BitVecs('a b c', 22)

s=Solver()

def bytes_in_UTF8_seq(x):
    if (x>>7)==0:
        return 1
    if (x>>5)==0b110:
        return 2
    if (x>>4)==0b1110:
        return 3
    if (x>>3)==0b11110:
        return 4
    # invalid 1st byte
    return None

for x in range(256):
    t=bytes_in_UTF8_seq(x)
    if t!=None:
        s.add(((a >> ((x>>b) & c)) & 3) == (t-1))

# enumerate all solutions:
results=[]
while s.check() == sat:
    m = s.model()

    print ("a,b,c = 0x%x 0x%x 0x%x" % (m[a].as_long(), m[b].as_long(), m[c].as_long()
       ))

    results.append(m)
    block = []
    for d in m:
        t=d()
        block.append(t != m[d])
    s.add(Or(block))

print ("results total=", len(results))
```

I tried various bit widths for a, b and c and found that 22 bits are enough. I've lots of results like:

```
...

a,b,c = 0x250100 0x3 0x381416
a,b,c = 0x258100 0x3 0x381416
a,b,c = 0x258900 0x3 0x381416
a,b,c = 0x250900 0x3 0x381416
a,b,c = 0x251100 0x3 0x381416
a,b,c = 0x259100 0x3 0x381416
a,b,c = 0x259100 0x3 0x389416
a,b,c = 0x251100 0x3 0x389416
a,b,c = 0x251100 0x3 0x189416
a,b,c = 0x259100 0x3 0x189416
a,b,c = 0x259100 0x3 0x189016

...
```

It seems that several least significant bits of a and c are not used. After little experimentation, I've come to this:

```
...

# make a,c more aesthetically appealing:
s.add((a&0xffff)==0)
s.add((c&0xffff00)==0)

...
```

And the results:

```
a,b,c = 0x250000 0x3 0x36
a,b,c = 0x250000 0x3 0x16
a,b,c = 0x250000 0x3 0x96
a,b,c = 0x250000 0x3 0xd6
a,b,c = 0x250000 0x3 0xf6
```

```
a,b,c = 0x250000 0x3 0x76
a,b,c = 0x250000 0x3 0xb6
a,b,c = 0x250000 0x3 0x56
results total= 8
```

Pick any.

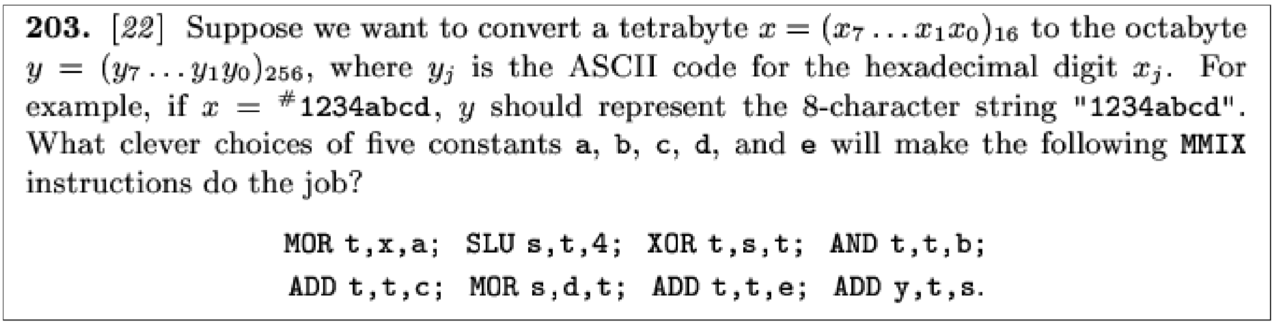
But how it works? Its operation is very similar to the bitwise trick related to leading/trailing zero bits counting based on De Bruijn sequences. Read more about it in Untitled collection of math notes[4].

The problem is small enough to be tackled by MK85: https://smt.st/current_tree/synth/pgm/TAOCP_713_198/TAOCP_713_198_MK85.py

Bruteforce is feasible here.

15.2.4 TAOCP 7.1.3 Exercise 203, MMIX MOR instruction and program synthesis by sketching

Found this exercise in TAOCP 7.1.3 (Bitwise Tricks and Techniques):



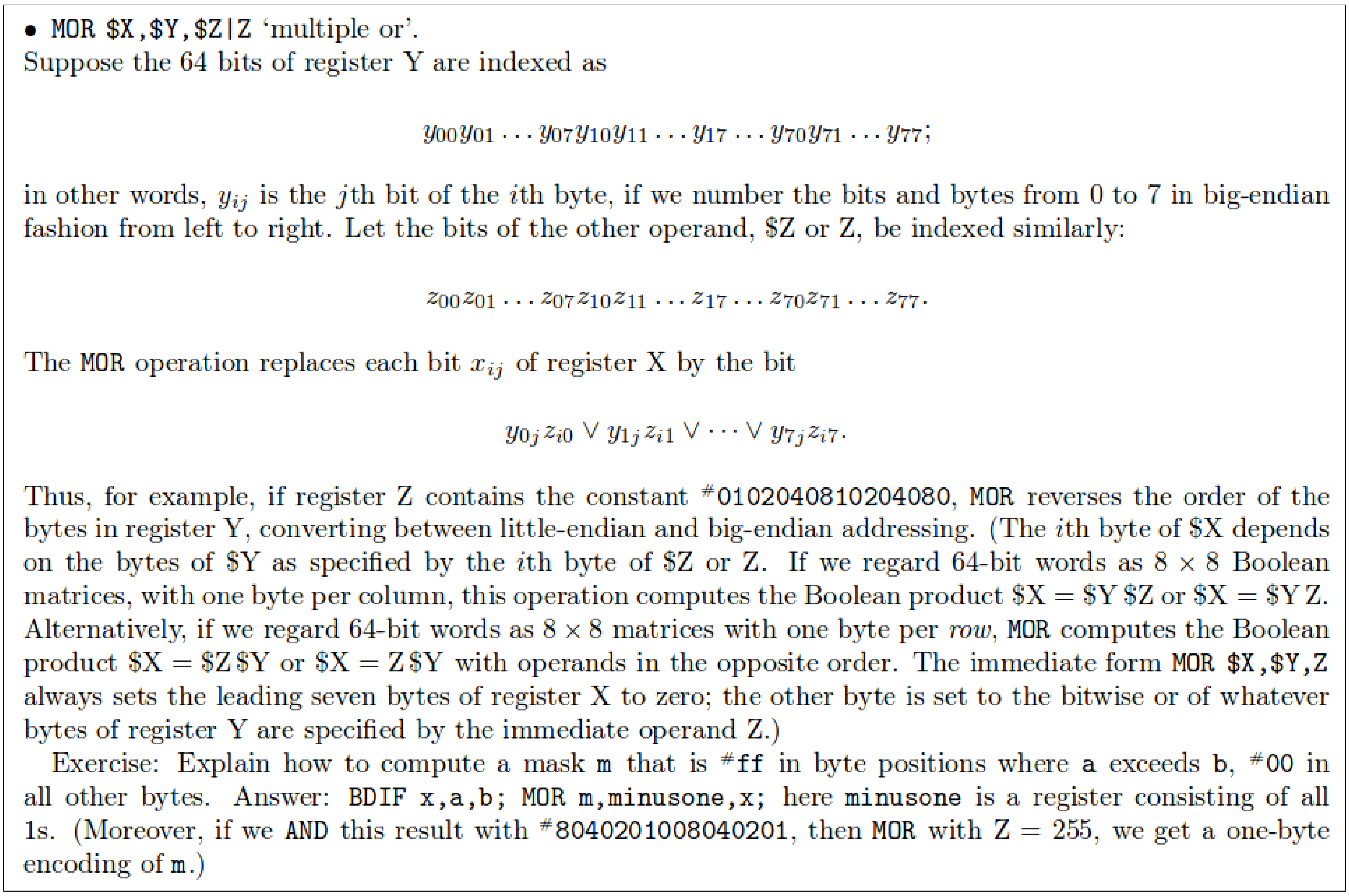
203. [*22*] Suppose we want to convert a tetrabyte $x = (x_7 \ldots x_1 x_0)_{16}$ to the octabyte $y = (y_7 \ldots y_1 y_0)_{256}$, where y_j is the ASCII code for the hexadecimal digit x_j. For example, if $x = $ #`1234abcd`, y should represent the 8-character string `"1234abcd"`. What clever choices of five constants `a`, `b`, `c`, `d`, and `e` will make the following `MMIX` instructions do the job?

```
MOR t,x,a;  SLU s,t,4;  XOR t,s,t;  AND t,t,b;
ADD t,t,c;  MOR s,d,t;  ADD t,t,e;  ADD y,t,s.
```

Figure 15.7: Screenshot from TAOCP book

What is MOR instruction in MMIX?

[4] https://xn--mxacd.xn--qxam/math-notes.pdf



• MOR $X,$Y,$Z|Z 'multiple or'.

Suppose the 64 bits of register Y are indexed as

$$y_{00}y_{01}\ldots y_{07}y_{10}y_{11}\ldots y_{17}\ldots y_{70}y_{71}\ldots y_{77};$$

in other words, y_{ij} is the jth bit of the ith byte, if we number the bits and bytes from 0 to 7 in big-endian fashion from left to right. Let the bits of the other operand, $Z or Z, be indexed similarly:

$$z_{00}z_{01}\ldots z_{07}z_{10}z_{11}\ldots z_{17}\ldots z_{70}z_{71}\ldots z_{77}.$$

The MOR operation replaces each bit x_{ij} of register X by the bit

$$y_{0j}z_{i0} \vee y_{1j}z_{i1} \vee \cdots \vee y_{7j}z_{i7}.$$

Thus, for example, if register Z contains the constant #0102040810204080, MOR reverses the order of the bytes in register Y, converting between little-endian and big-endian addressing. (The ith byte of $X depends on the bytes of $Y as specified by the ith byte of $Z or Z. If we regard 64-bit words as 8 × 8 Boolean matrices, with one byte per column, this operation computes the Boolean product $X = $Y $Z or $X = $Y Z. Alternatively, if we regard 64-bit words as 8 × 8 matrices with one byte per *row*, MOR computes the Boolean product $X = $Z $Y or $X = Z $Y with operands in the opposite order. The immediate form MOR $X,$Y,Z always sets the leading seven bytes of register X to zero; the other byte is set to the bitwise or of whatever bytes of register Y are specified by the immediate operand Z.)

Exercise: Explain how to compute a mask m that is #ff in byte positions where a exceeds b, #00 in all other bytes. Answer: BDIF x,a,b; MOR m,minusone,x; here minusone is a register consisting of all 1s. (Moreover, if we AND this result with #8040201008040201, then MOR with Z = 255, we get a one-byte encoding of m.)

Figure 15.8: Screenshot from the MMIX book

(http://mmix.cs.hm.edu/doc/mmix-doc.pdf)

Let's try to solve. We create two functions. First has MOR instructions simulation + the program from TAOCP. The second is a naive implementation. Then we add "forall" quantifier: for all inputs, both functions must produce the same result. **But**, we don't know a/b/c/d/e/f and ask Z3 SMT-solver to

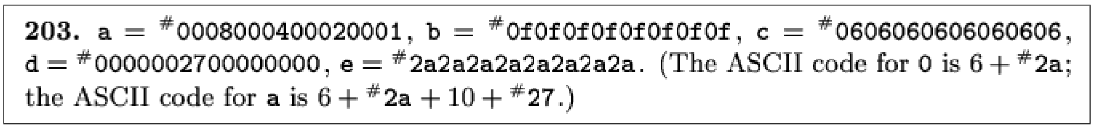
```
#!/usr/bin/env python3
from z3 import *

s=Solver()
set_param("parallel.enable", True)

a, b, c, d, e=BitVecs('a b c d e', 64)

def simulate_MOR(y,z):
    """
    set each bit of 64-bit result, as:

    $x_i_j = y_0_j z_i_0 \vee y_1_j z_i_1 \vee \cdots \vee y_7_j z_i_7$
    https://latexbase.com/d/bf2243f8-5d0b-4231-8891-66fb47d846f0

    IOW:

    x<byte><bit> = (y<0><bit> AND z<byte><0>) OR (y<1><bit> AND z<byte><1>) OR ... OR
        (y<7><bit> AND z<byte><7>)
    """

    def get_ij(x, i, j):
        return (x>>(i*8+j))&1
    rt=0
    for byte in range(8):
        for bit in range(8):
            t=0
            for i in range(8):
                t|=get_ij(y, i, bit) & get_ij(z, byte, i)

            pos=byte*8+bit
            rt|=t<<pos

    return rt

def simulate_pgm(x):
    t=simulate_MOR(x,a)
    s=t<<4
    t=s^t
    t=t&b
    t=t+c
    s=simulate_MOR(d,t)
    t=t+e
    y=t+s
    return y

def nibble_to_ASCII(x):
    return If(And(x>=0, x<=9), 0x30+x, 0x61+x-10)

def method2(x):
    rt=0
    for i in range(8):
        rt|=nibble_to_ASCII((x >> i*4)&0xf) << i*8
    return rt

#"""
# new version.
# for all possible 32-bit x's, find such a/b/c/d/e, so that these two parts would be equal to each other
# zero extend x to 64-bit value in both cases
x=BitVec('x', 32)
s.add(ForAll([x], simulate_pgm(ZeroExt(32, x))==method2(ZeroExt(32, x))))
#"""

"""
# previous version:
for i in range(5):
    x=random.getrandbits(32)
    t="%08x" % x
    y=int(''.join("%02X" % ord(c) for c in t), 16)
    print ("%x %x" % (x, y))

    s.add(simulate_pgm(x)==y)
"""

# enumerate all solutions:
results=[]
while s.check() == sat:
    m = s.model()

    print ("a,b,c,d,e = %x %x %x %x %x" % (m[a].as_long(), m[b].as_long(), m[c].as_long(), m[d].as_long(), m[e].as_long()))

    results.append(m)
    block = []
    for d1 in m:
        t=d1()
        block.append(t != m[d1])
    s.add(Or(block))

print ("results total=", len(results))
```

Very slow, it takes 30m on Intel Quad-Core Xeon E3-1270 v3 3.50GHz but found at least one solution:

```
a,b,c,d,e = 8000400020001 f0f0f0f0f0f0f0f 56d656d616969616 411a00000000
    bf3fbf3fff7f8000
```

...which is correct (I've wrote bruteforce checker, here: https://smt.st/current_tree/synth/pgm/TAOCP_713_203/check.c.

D.Knuth's TAOCP also has answers:



203. a = #0008000400020001, b = #0f0f0f0f0f0f0f0f, c = #0606060606060606, d = #0000002700000000, e = #2a2a2a2a2a2a2a2a. (The ASCII code for 0 is 6 + #2a; the ASCII code for a is 6 + #2a + 10 + #27.)

Figure 15.9: Screenshot from TAOCP book

...which are different, but also correct.

What if a==0x0008000400020001 always? I'm adding a new constraint:

```
s.add(a==0x0008000400020001)
```

We've getting many results (much faster, and also correct):

```
...

a,b,c,d,e = 8000400020001 7f0fcf0fcf0f7f0f 1616d6d656561656 8680522903020000
    eeeda9aa2e2eee2f
a,b,c,d,e = 8000400020001 7f0fcf0fcf0f6f0f 1616d6d656561656 8680522903020000
    eeeda9aa2e2eee2f
a,b,c,d,e = 8000400020001 7f0fcf0fdf0f6f0f 1616d6d656561656 8680522903020000
    eeeda9aa2e2eee2f
a,b,c,d,e = 8000400020001 5f0fcf0fdf0f6f0f 1616d6d656561656 8680522903020000
    eeeda9aa2e2eee2f

...
```

The files: https://smt.st/current_tree/synth/pgm/TAOCP_713_203

Bruteforce wouldn't be feasible, unless you'll guide it using heuristics.

15.2.5 Loading a constant into register using ASCII-only x86 code

... this is a task often required when constructing shellcodes. I'm not sure if this is still relevant these days, however, it was fun to do it.

I've picked 3 instructions with ASCII-only opcodes:

```
26 25 xx xx xx xx    and     eax, imm32
26 2D xx xx xx xx    sub     eax, imm32
26 35 xx xx xx xx    xor     eax, imm32
```

Will it be possible to generate such a sequence of instructions, so that the arbitrary 32-bit constant would be loaded into EAX register? Given the fact that the initial value of EAX is unknown, because, let's say, we can't reset it? Surely, all 32-bit operands must have ASCII-only bytes as well.

The answer is... using Z3 SMT-solver:

```
#!/usr/bin/env python3

from z3 import *
```

```
import sys, random

BIT_WIDTH=32

MAX_STEPS=20

#CONST=0
#CONST=0x12345678
#CONST=0x0badf00d
#CONST=0xffffffff

CONST=random.randint(0,0x100000000)
print ("CONST=0x%x" % CONST)

CHAINS=30

def simulate_op(R, c, op, op1_val, op2_imm, STEPS):
    return If(op==0, op1_val - op2_imm,
           If(op==1, op1_val ^ op2_imm,
           If(op==2, op1_val & op2_imm,
               0))) # default

op_to_str_tbl=["SUB", "XOR", "AND"]
instructions=len(op_to_str_tbl)

def print_model(m, STEPS, op, op2_imm):
    for s in range(1,STEPS):
        op_n=m[op[s]].as_long()
        op_s=op_to_str_tbl[op_n]
        print ("%s EAX, 0x%x" % (op_s, m[op2_imm[s]].as_long()))

def attempt(STEPS):
    print ("attempt, STEPS=", STEPS)
    sl=Solver()

    R=[[BitVec('S_s%d_c%d' % (s, c), BIT_WIDTH) for s in range(MAX_STEPS)] for c in
       range (CHAINS)]
    op=[Int('op_s%d' % s) for s in range(MAX_STEPS)]
    op2_imm=[BitVec('op2_imm_s%d' % s, BIT_WIDTH) for s in range(MAX_STEPS)]

    for s in range(1, STEPS):
        # for each step, instruction is in 0..2 range:
        sl.add(Or(op[s]==0, op[s]==1, op[s]==2))

        # each 8-bit byte in operand must be in [0x21..0x7e] range:
        # or 0x20, if space character is tolerated...
        for shift_cnt in [0,8,16,24]:
            sl.add(And((op2_imm[s]>>shift_cnt)&0xff>=0x21),(op2_imm[s]>>shift_cnt)&0
               xff<=0x7e)

        """
        # or use 0..9, a..z, A..Z:
        for shift_cnt in [0,8,16,24]:
            sl.add(Or(
                And((op2_imm[s]>>shift_cnt)&0xff>=ord('0'),(op2_imm[s]>>shift_cnt)&0
                   xff<=ord('9')),
                And((op2_imm[s]>>shift_cnt)&0xff>=ord('a'),(op2_imm[s]>>shift_cnt)&0
                   xff<=ord('z')),
                And((op2_imm[s]>>shift_cnt)&0xff>=ord('A'),(op2_imm[s]>>shift_cnt)&0
                   xff<=ord('Z'))))
        """
    # for all input random numbers, the result must be CONST:
    for c in range(CHAINS):
        sl.add(R[c][0]==random.randint(0,0x100000000))
        sl.add(R[c][STEPS-1]==CONST)

        for s in range(1, STEPS):
            sl.add(R[c][s]==simulate_op(R,c, op[s], R[c][s-1], op2_imm[s], STEPS))

    tmp=sl.check()
    if tmp==sat:
        print ("sat!")
        m=sl.model()
        print_model(m, STEPS, op, op2_imm)
        exit(0)
    else:
        print (tmp)

for s in range(2, MAX_STEPS):
    attempt(s)
```

What it can generate for zero:

```
AND EAX, 0x3e5a3e28
AND EAX, 0x40214040
```

These two instructions clears EAX. You can understand how it works if you'll see these operands in binary form:

```
0x3e5a3e28 =     111110010110100011111000101000
0x40214040 =    1000000001000010100000001000000
```

It's best to have a zero bit for both operands, but this is not always possible, because each of 4 bytes in 32-bit operand must be in [0x21..0x7e] range, so Z3 solver find a way to reset other bits using second instruction.

Running it again:

```
AND EAX, 0x3c5e3621
AND EAX, 0x42214850
```

Operands are different, because SAT solver is probably initialized randomly.

Now 0x0badf00d:

```
AND EAX, 0x48273048
AND EAX, 0x31504325
XOR EAX, 0x61212251
SUB EAX, 0x55733244
```

First two AND instruction clears EAX, 3th and 4th makes 0x0badf00d value.

Now 0x12345678:

```
AND EAX, 0x41212230
XOR EAX, 0x292f2224
AND EAX, 0x365e4048
XOR EAX, 0x323a5678
```

Slightly different, but also correct.

For some constants, more instructions required:

```
CONST=0xf3c37766
...
AND EAX, 0x21283024
AND EAX, 0x58504050
SUB EAX, 0x31377b56
SUB EAX, 0x3f2f3b5e
XOR EAX, 0x7c5a3e2a
```

Now what if, for aesthetical reasons maybe, we would limit all printable characters to 0..9, a..z, A..Z (comment/uncomment corresponding fragments of the source code)? This is not a problem at all.

However, if to limit to a..z, A..Z, it needs more instructions, but this is still correct (8 instructions to clear EAX register):

```
CONST=0x0
...
XOR EAX, 0x43685575
SUB EAX, 0x6c747a6f
XOR EAX, 0x59525541
AND EAX, 0x65755454
XOR EAX, 0x57416643
AND EAX, 0x76767757
SUB EAX, 0x556f7547
AND EAX, 0x42424242
```

However, 7 instructions for 0x12345678 constant:

```
CONST=0x12345678
...
AND EAX, 0x6f77414d
SUB EAX, 0x646b7845
AND EAX, 0x41674a54
SUB EAX, 0x47414744
AND EAX, 0x49486d41
XOR EAX, 0x53757778
AND EAX, 0x7274567a
```

Further work: use ForAll quantifier instead of randomly generated test inputs... also, we could try INC EAX/DEC EAX instructions.

What about bruteforce? Each instruction is at least 32 bits... Not feasible, unless you can find heuristics...

15.2.6 Further reading

”A toy code generator” https://github.com/nickgildea/z3_codegen — Nick Gildea has introduced a ”set” instruction, loading a value into register.

The “Dragon book” (Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman – Compilers. Principles, Techniques, and Tools), 2nd ed., 2006, has a part 6.2.3 Triples, which describes the same way of representing programs like I do here is this book.

15.3 DFA (Deterministic Finite Automaton) synthesis

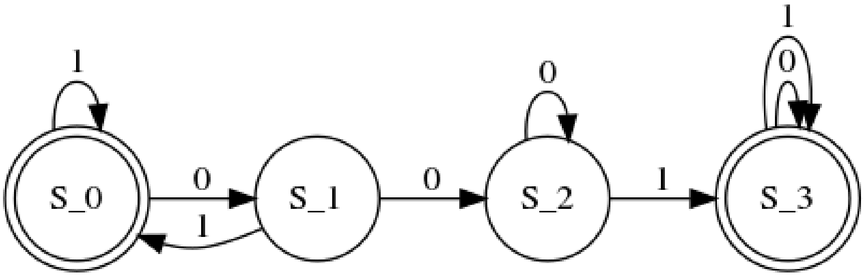
15.3.1 DFA accepting a 001 substring

```
This example shows how to design a finite automaton E_2 to recognize the regular
language of all strings that contain the string 001 as a substring. For example,
0010, 1001, 001, and 11111110011111 are all in the language, but 11 and 0000
are not. How would you recognize this language if you were pretending to
be E_2?
```

(M.Sipser — Introduction to the Theory of Computation, 2ed, p43, Example 1.21.)

```
#!/usr/bin/env python3
from z3 import *
import sys, random, os

state=[[Int('state_%d_%d' % (s, b)) for b in range(2)] for s in range(100)]

INITIAL_STATE=0
INVALID_STATE=999

# construct FA for z3
```

```
def transition (STATES, s, i):
    # this is like switch()

    rt=IntVal(INVALID_STATE)
    for j in range(STATES):
        rt=If(And(s==IntVal(j), i==0), state[j][0], rt)
        rt=If(And(s==IntVal(j), i==1), state[j][1], rt)
    return rt

# construct FA for z3
def FA(STATES, input_string):
    s=IntVal(INITIAL_STATE)
    for i in input_string:
        s=transition(STATES, s, int(i))
    return s

def print_model(STATES, m):
    print ("[state, input, new state]")
    for i in range(STATES):
        print ("[%d, \"0\", %d]," % (i, m[state[i][0]].as_long()))
        print ("[%d, \"1\", %d]," % (i, m[state[i][1]].as_long()))

    f=open("1.gv", "wt")
    f.write("digraph finite_state_machine {\n")
    f.write("\trankdir=LR;\n")
    f.write("\tsize=\"8,5\"\n")
    f.write("\tnode [shape = doublecircle]; S_0 S_"+str(STATES-1)+";\n");
    f.write("\tnode [shape = circle];\n");

    FA={}
    for s in range(STATES):
        f.write("\tS_%d -> S_%d [ label = \"0\" ];\n" % (s, m[state[s][0]].as_long())
           )
        f.write("\tS_%d -> S_%d [ label = \"1\" ];\n" % (s, m[state[s][1]].as_long())
           )
        FA[s]=(m[state[s][0]].as_long(), m[state[s][1]].as_long())
    f.write("}\n")
    f.close()
    os.system("dot -Tpng 1.gv > 1.png") # run GraphViz

def attempt(STATES):
    print ("STATES=", STATES)
    sl=Solver()
    # Z3 multithreading, starting at 4.8.x:
    set_param("parallel.enable", True)

    for s in range(STATES):
        for b in range(2):
            sl.add(And(state[s][b]>=0, state[s][b]<STATES))

    ACCEPTING_STATE=STATES-1

    for i in range(256):
        b=bin(i)[2:]
        if "001" in b:
            sl.add(FA(STATES, b)==ACCEPTING_STATE)
        else:
            sl.add(FA(STATES, b)!=ACCEPTING_STATE)

    result=[]
    if sl.check() == unsat:
        return
    m = sl.model()
    print_model(STATES, m)
    exit(0)

for i in range(2, 100):
    attempt(i)
```



15.3.2 DFA accepting a binary substring divisible by prime number

A problem: construct a regular expression accepting binary number divisible by 3.

”1111011” (123) is, ”101010101011” (2731) is not.

Some discussion and the correct expressions:

- https://stackoverflow.com/questions/7974655/regex-for-binary-multiple-of-3
- https://stackoverflow.com/questions/844867/check-if-a-number-is-divisible-by-3
- https://stackoverflow.com/questions/867279/regular-expression-to-define-some-binary-sequence
- https://www.regextester.com/96234

I couldn’t generate RE, but I can generate a minimal DFA:

```
#!/usr/bin/env python3
from z3 import *
import sys, random, os

state=[[Int('state_%d_%d' % (s, b)) for b in range(2)] for s in range(100)]

INITIAL_STATE=0
INVALID_STATE=999

# construct FA for z3
def transition (STATES, s, i):
    # this is like switch()

    rt=IntVal(INVALID_STATE)
    for j in range(STATES):
        rt=If(And(s==IntVal(j), i==0), state[j][0], rt)
        rt=If(And(s==IntVal(j), i==1), state[j][1], rt)
    return rt

# construct FA for z3
def FA(STATES, input_string):
    s=IntVal(INITIAL_STATE)
    for i in input_string:
        s=transition(STATES, s, int(i))
    return s
```

```
# simulate FA for testing purpose:
def simulate_FA(input_, FA):
    s=INITIAL_STATE
    for i in input_:
        if i=='0':
            s=FA[s][0]
        else:
            s=FA[s][1]
    return s

def test_FA (DIVISOR, STATES, FA):
    ACCEPTING_STATE=STATES-1
    for i in range(10000):
        rnd=random.randint(1,100000000000000)
        b=bin(rnd)[2:]
        final_state=simulate_FA(b, FA)
        if (rnd % DIVISOR)==0:
            if final_state!=ACCEPTING_STATE:
                print ("error! FA is invalid")
                exit(0)
        else:
            if final_state==ACCEPTING_STATE:
                print ("error! FA is invalid")
                exit(0)
    print ("test OK")

def print_model(DIVISOR, STATES, m):
    print ("[state, input, new state]")
    for i in range(STATES):
        print ("[%d, \"0\", %d]," % (i, m[state[i][0]].as_long()))
        print ("[%d, \"1\", %d]," % (i, m[state[i][1]].as_long()))

    fname_base="DIVISOR_%d_STATES_%d" % (DIVISOR, STATES)
    f=open(fname_base+".gv", "wt")
    f.write("digraph finite_state_machine {\n")
    f.write("\trankdir=LR;\n")
    f.write("\tsize=\"8,5\"\n")
    f.write("\tnode [shape = doublecircle]; S_0 S_"+str(STATES-1)+";\n");
    f.write("\tnode [shape = circle];\n");

    FA={}
    for s in range(STATES):
        f.write("\tS_%d -> S_%d [ label = \"0\" ];\n" % (s, m[state[s][0]].as_long())
            )
        f.write("\tS_%d -> S_%d [ label = \"1\" ];\n" % (s, m[state[s][1]].as_long())
            )
        FA[s]=(m[state[s][0]].as_long(), m[state[s][1]].as_long())
    f.write("}\n")
    f.write("// STATES=%d\n" % STATES)
    f.close()
    os.system("dot -Tpng "+fname_base+".gv > "+fname_base+".png") # run GraphViz
    test_FA (DIVISOR, STATES, FA)

def attempt(DIVISOR, STATES):
    print ("DIVISOR=%d STATES=%d" % (DIVISOR, STATES))
    sl=Solver()
    # Z3 multithreading, starting at 4.8.x:
    set_param("parallel.enable", True)
    #sl.set("timeout", 60*60) # 1h

    for s in range(STATES):
        for b in range(2):
            sl.add(And(state[s][b]>=0, state[s][b]<STATES))

    ACCEPTING_STATE=STATES-1

    # may be lower for low DIVISOR's like 3 or 5.
    # but 256 is safe choice for DIVISOR's up to 9
    #for i in range(256):
    for i in range(512):
        b=bin(i)[2:]
        if (i % DIVISOR)==0:
            sl.add(FA(STATES, b)==ACCEPTING_STATE)
        else:
            sl.add(FA(STATES, b)!=ACCEPTING_STATE)

    result=[]

    if sl.check() == unsat:
        print ("unsat")
        return False
    if sl.check() == unknown:
        print ("unknown")
        return False
    m = sl.model()
    print_model(DIVISOR, STATES, m)
    return True

DIVISOR=int(sys.argv[1])
STATES=int(sys.argv[2])
attempt(DIVISOR, STATES)
```

As you can see, it has testing procedure, which is, in turn, can be used instead of RE matcher, if you really need to match numbers divisible by 3.

States in double circles — initial (S_0) and accepting:

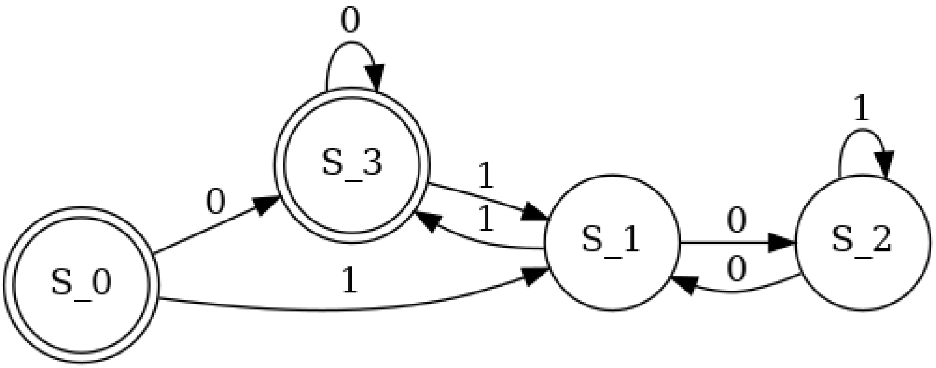


Figure 15.10: DFA for numbers divisible by 3 (4 states) (minimal)

... is almost like the one someone posted here, but my solution has two separate states as initial and accepting.

Some of solutions are hard to find manually.

DFAs are bigger for prime numbers, of course, and smaller for even numbers.

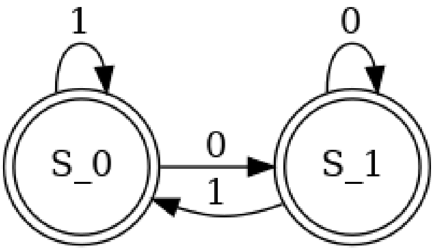


Figure 15.11: DFA for numbers divisible by 2 (2 states) (minimal)

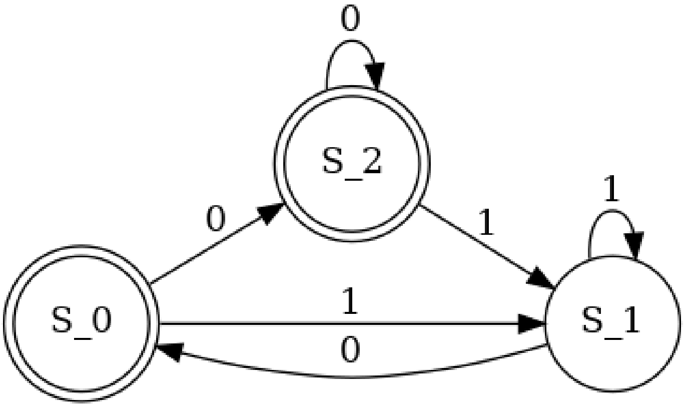


Figure 15.12: DFA for numbers divisible by 4 (3 states) (minimal)

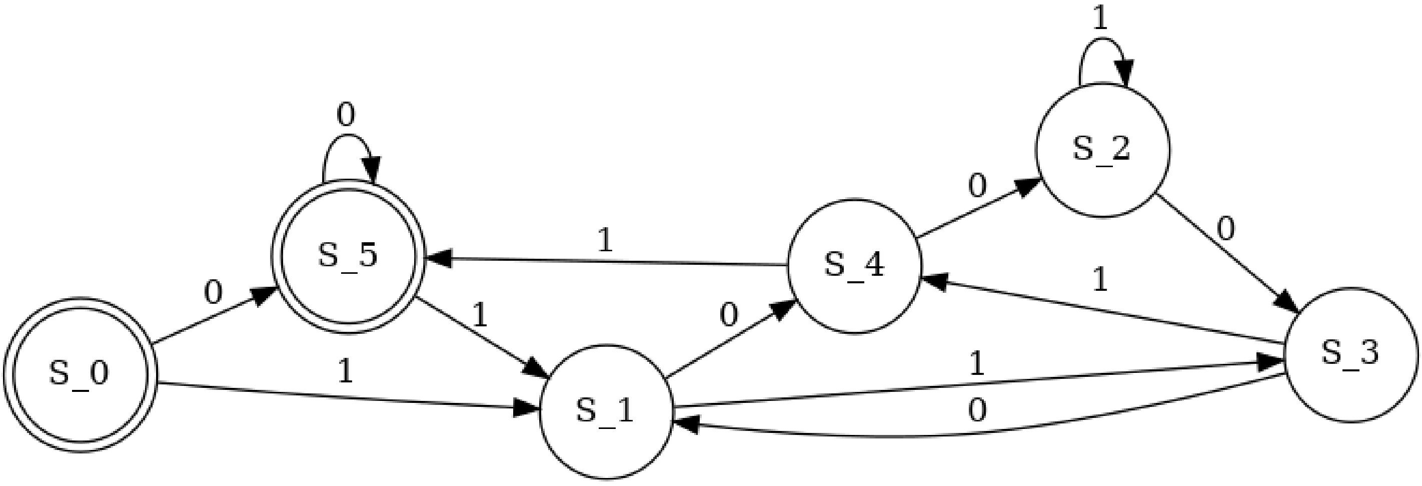


Figure 15.13: DFA for numbers divisible by 5 (6 states) (minimal)

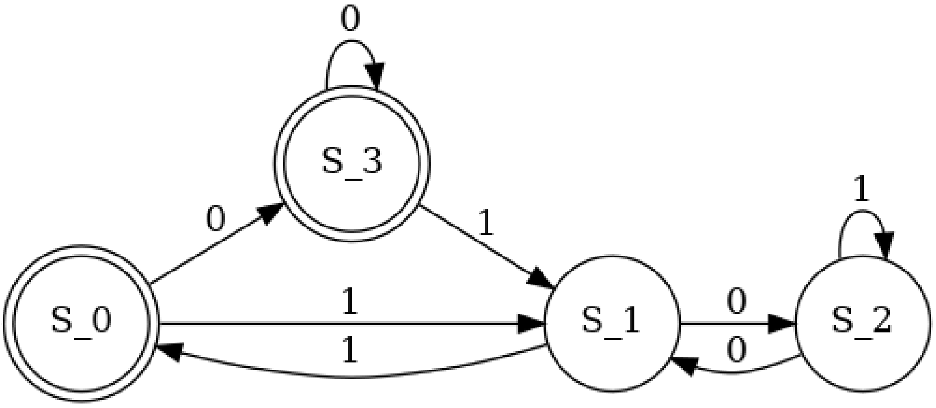


Figure 15.14: DFA for numbers divisible by 6 (4 states) (minimal)

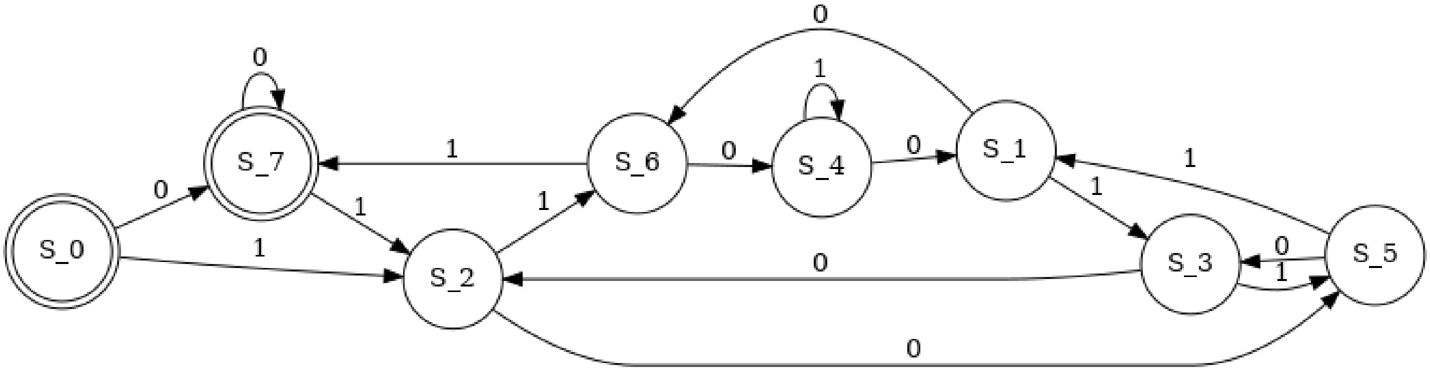


Figure 15.15: DFA for numbers divisible by 7 (8 states) (minimal)

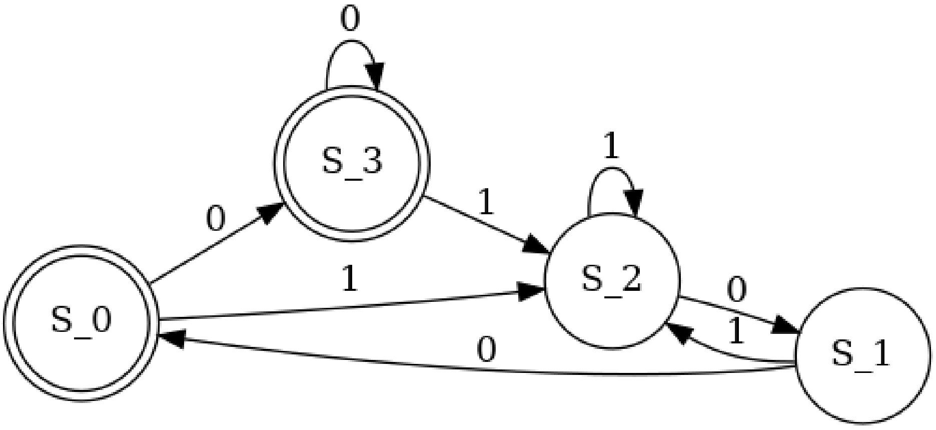


Figure 15.16: DFA for numbers divisible by 8 (4 states) (minimal)

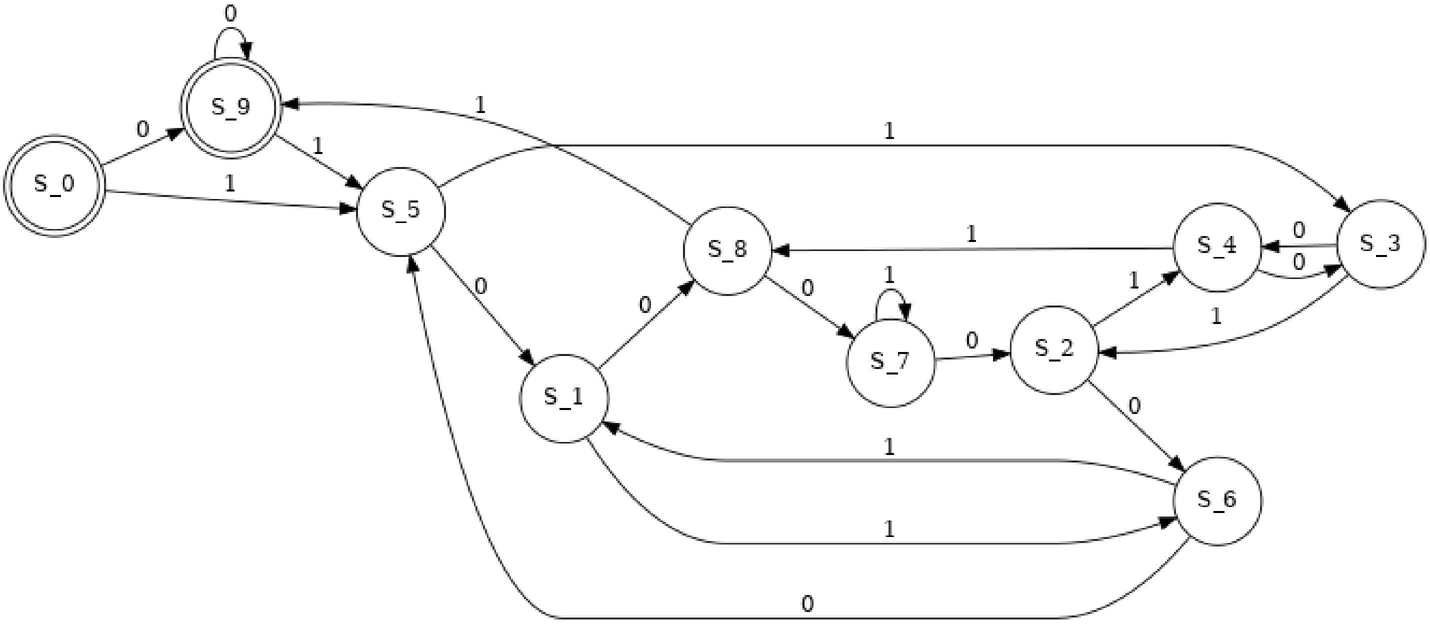


Figure 15.17: DFA for numbers divisible by 9 (10 states) (minimal)

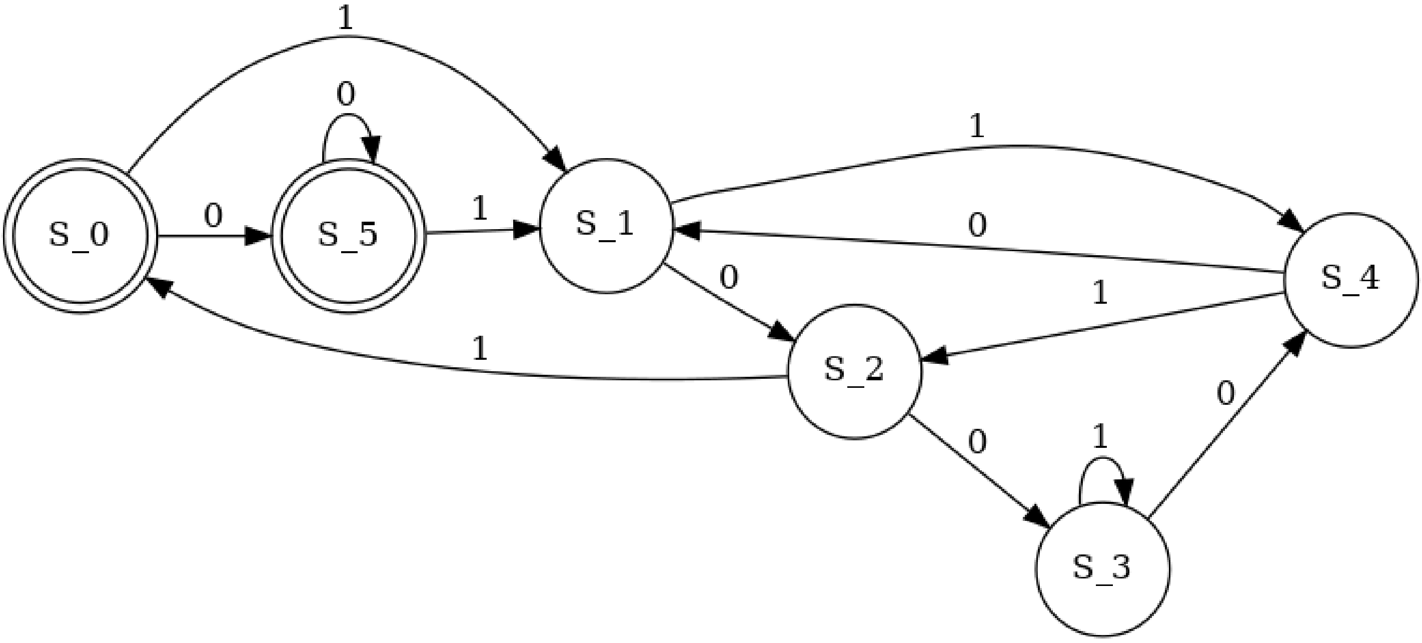


Figure 15.18: DFA for numbers divisible by 10 (6 states) (minimal)

These DFAs above are guaranteed to be minimal.

The following DFA may not be minimal. I couldn't find it for 11 states after Z3 working for one hour, but I could find it for 12 states. So it may be minimal, but may be not.

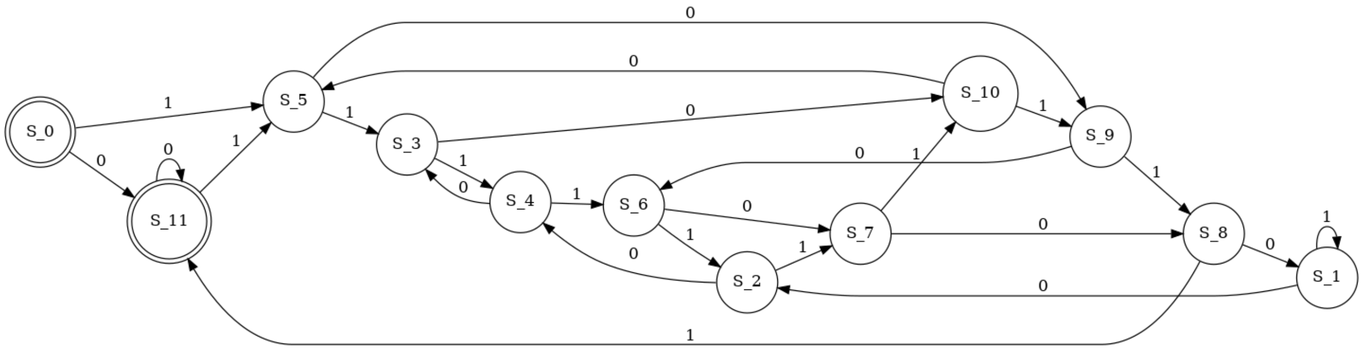


Figure 15.19: DFA for numbers divisible by 11 (12 states) (may be not minimal)

But this is minimal:

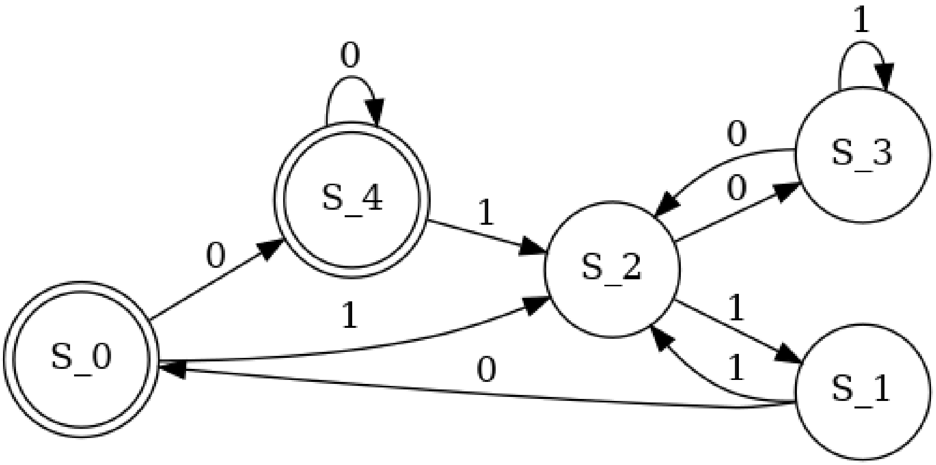


Figure 15.20: DFA for numbers divisible by 12 (5 states) (minimal)

May not be minimal again. Search failed for 11, 12, 13 states (one hour for each), but found for 14 states:

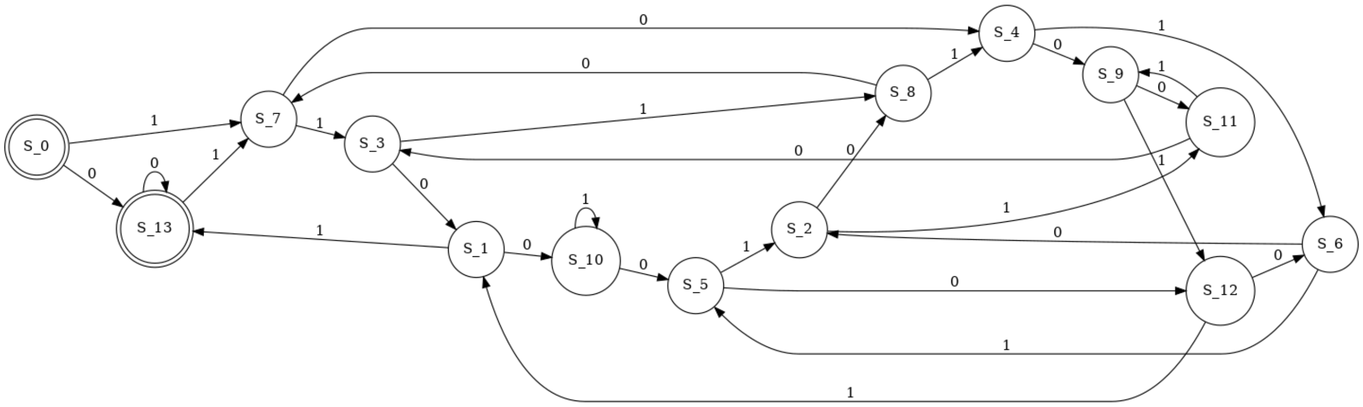


Figure 15.21: DFA for numbers divisible by 13 (14 states) (may be not minimal)

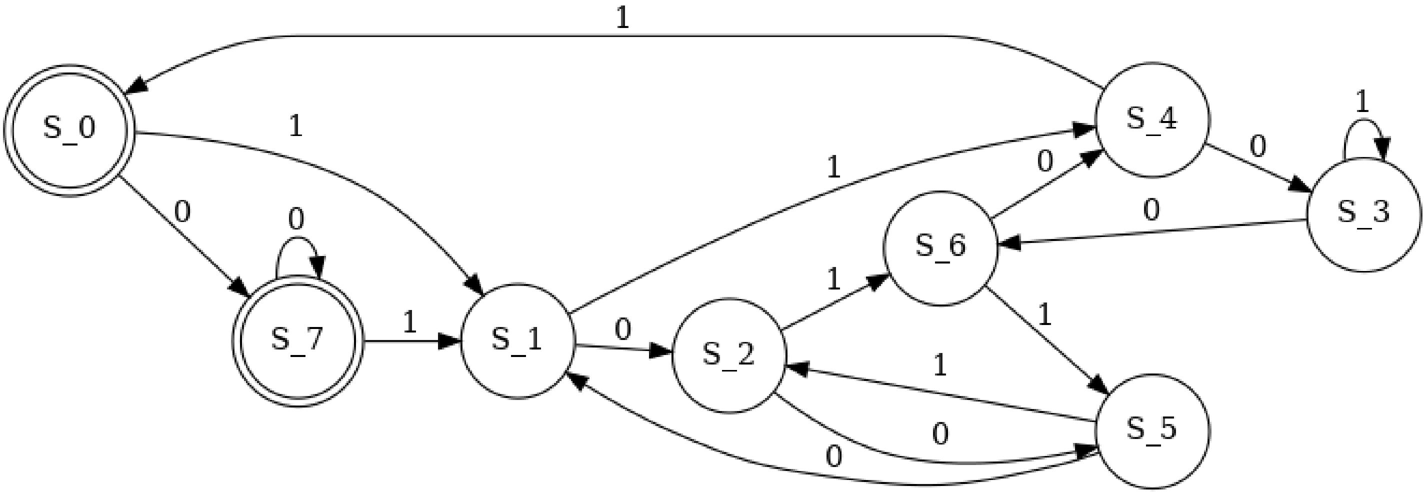


Figure 15.22: DFA for numbers divisible by 14 (8 states) (minimal)

May be not be minimal. Search failed for 10..15 states, but found for 16 states:

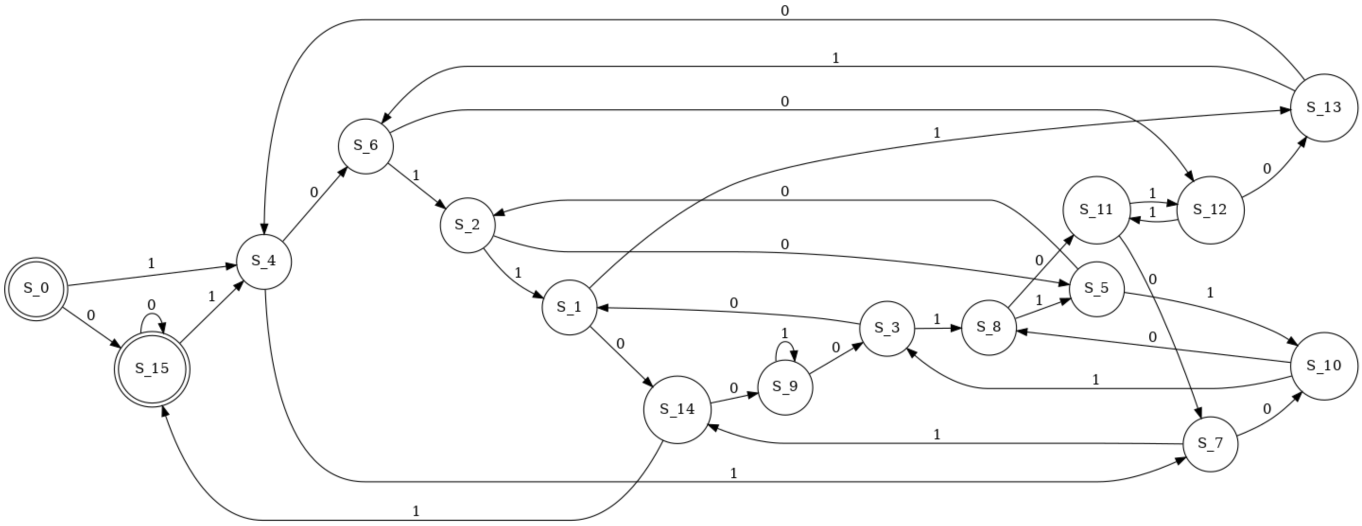


Figure 15.23: DFA for numbers divisible by 15 (16 states) (may be not minimal)

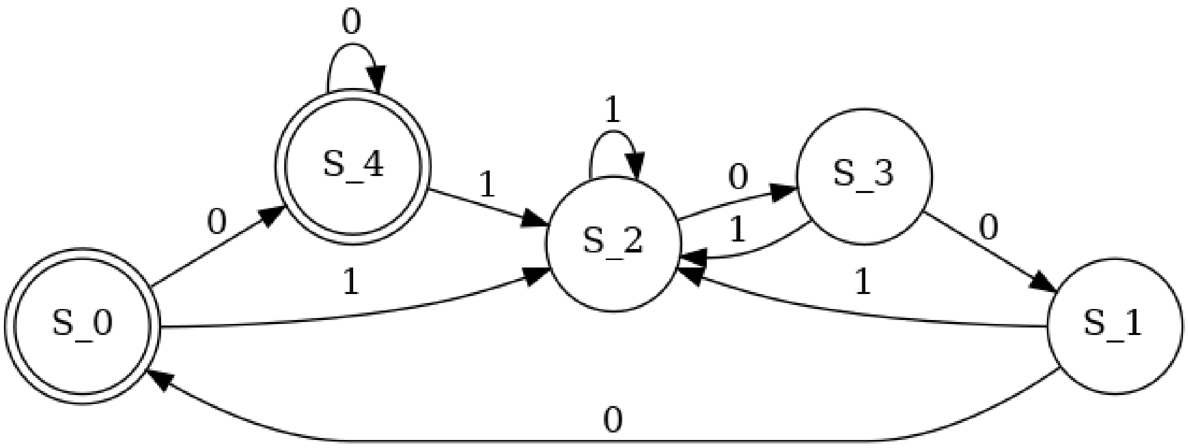


Figure 15.24: DFA for numbers divisible by 16 (5 states) (minimal)

Further work

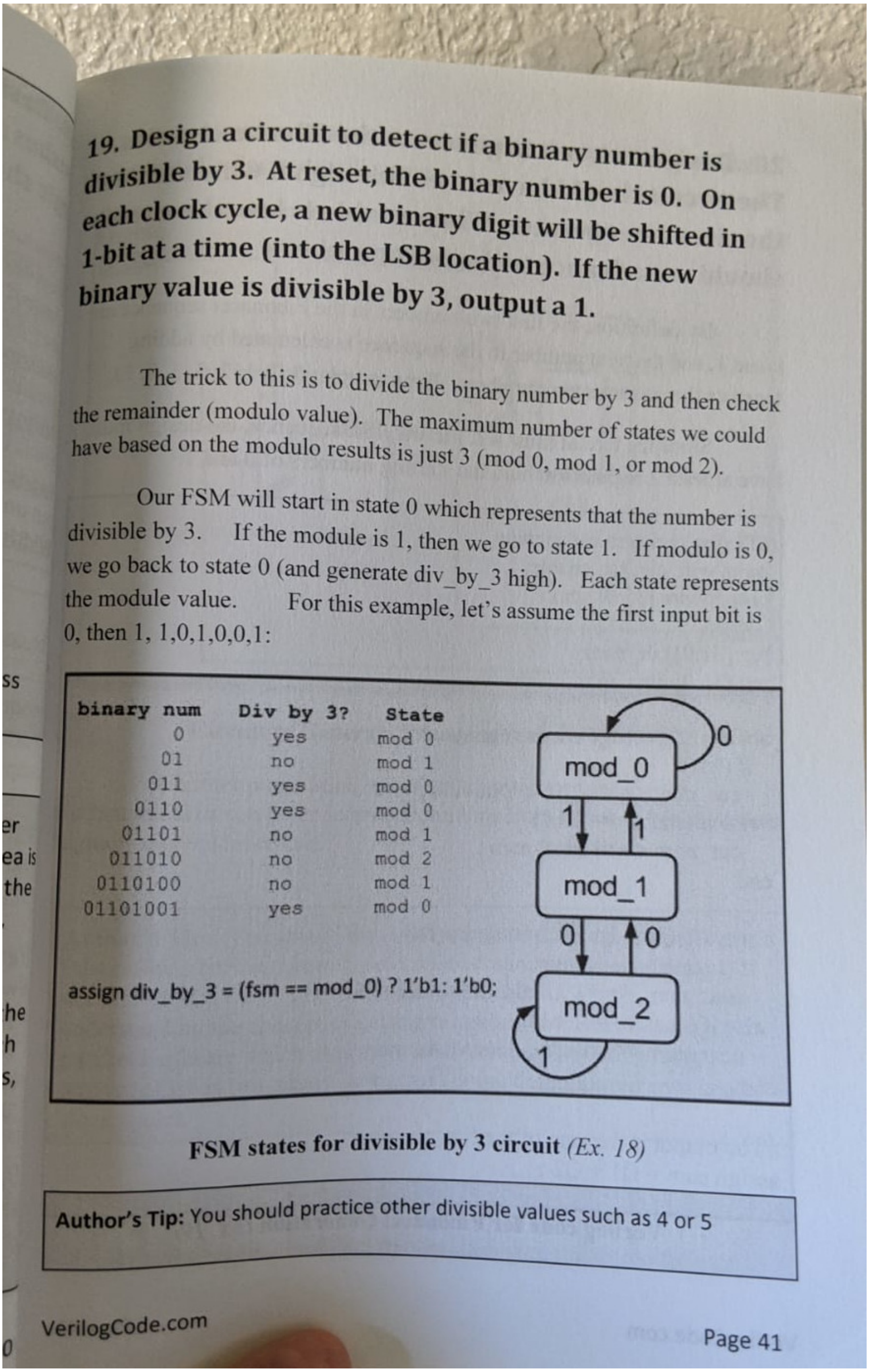
Convert DFAs to RE...

What about bruteforce?

For 10 vertices, you have to enumerate $(10 \cdot 10)^{10} = 10^{20}$ DFAs, or $log_2(10 \cdot 10)^{10} \approx 66$ bits.

The files

https://smt.st/current_tree/synth/DFA



19. Design a circuit to detect if a binary number is divisible by 3. At reset, the binary number is 0. On each clock cycle, a new binary digit will be shifted in 1-bit at a time (into the LSB location). If the new binary value is divisible by 3, output a 1.

The trick to this is to divide the binary number by 3 and then check the remainder (modulo value). The maximum number of states we could have based on the modulo results is just 3 (mod 0, mod 1, or mod 2).

Our FSM will start in state 0 which represents that the number is divisible by 3. If the module is 1, then we go to state 1. If modulo is 0, we go back to state 0 (and generate div_by_3 high). Each state represents the module value. For this example, let's assume the first input bit is 0, then 1, 1,0,1,0,0,1:

binary num	Div by 3?	State
0	yes	mod 0
01	no	mod 1
011	yes	mod 0
0110	yes	mod 0
01101	no	mod 1
011010	no	mod 2
0110100	no	mod 1
01101001	yes	mod 0

assign div_by_3 = (fsm == mod_0) ? 1'b1: 1'b0;

FSM states for divisible by 3 circuit *(Ex. 18)*

Author's Tip: You should practice other divisible values such as 4 or 5

VerilogCode.com

Page 41

Figure 15.25: A problem from the "Digital Logic RTL & Verilog Interview Questions" book

I've found the problem here (Russian Facebook post). The book is available at amazon[5]. More information on its website.

15.3.3 Further reading

https://en.wikipedia.org/wiki/Induction_of_regular_languages

15.3.4 Food for thought

There is a lot of information about divisibility tests in decimal base: https://mathworld.wolfram.com/DivisibilityTests.html.

15.4 Other

15.4.1 Graph theory: degree sequence problem / graph realization problem

> The degree sequence of an undirected graph is the non-increasing sequence of its vertex degrees;[2] for the above graph it is (5, 3, 3, 2, 2, 1, 0). The degree sequence is a graph invariant so isomorphic graphs have the same degree sequence. However, the degree sequence does not, in general, uniquely identify a graph; in some cases, non-isomorphic graphs have the same degree sequence.
>
> The degree sequence problem is the problem of finding some or all graphs with the degree sequence being a given non-increasing sequence of positive integers. (Trailing zeroes may be ignored since they are trivially realized by adding an appropriate number of isolated vertices to the graph.) A sequence which is the degree sequence of some graph, i.e. for which the degree sequence problem has a solution, is called a graphic or graphical sequence. As a consequence of the degree sum formula, any sequence with an odd sum, such as (3, 3, 1), cannot be realized as the degree sequence of a graph. The converse is also true: if a sequence has an even sum, it is the degree sequence of a multigraph. The construction of such a graph is straightforward: connect vertices with odd degrees in pairs by a matching, and fill out the remaining even degree counts by self-loops. The question of whether a given degree sequence can be realized by a simple graph is more challenging. This problem is also called graph realization problem and can either be solved by the Erdős–Gallai theorem or the Havel–Hakimi algorithm. The problem of finding or estimating the number of graphs with a given degree sequence is a problem from the field of graph enumeration.

(https://en.wikipedia.org/wiki/Degree_(graph_theory))

```
The graph realization problem is a decision problem in graph theory. Given a finite
    sequence ( d 1 , … , d n )
{\displaystyle (d_{1},\dots ,d_{n})} {\displaystyle (d_{1},\dots ,d_{n})} of natural
    numbers, the problem asks whether there is
labeled simple graph such that ( d 1 , … , d n ) {\displaystyle (d_{1},\dots ,d_{n})}
     {\displaystyle (d_{1},\dots ,d_{n})}
is the degree sequence of this graph.
```

(https://en.wikipedia.org/wiki/Graph_realization_problem)

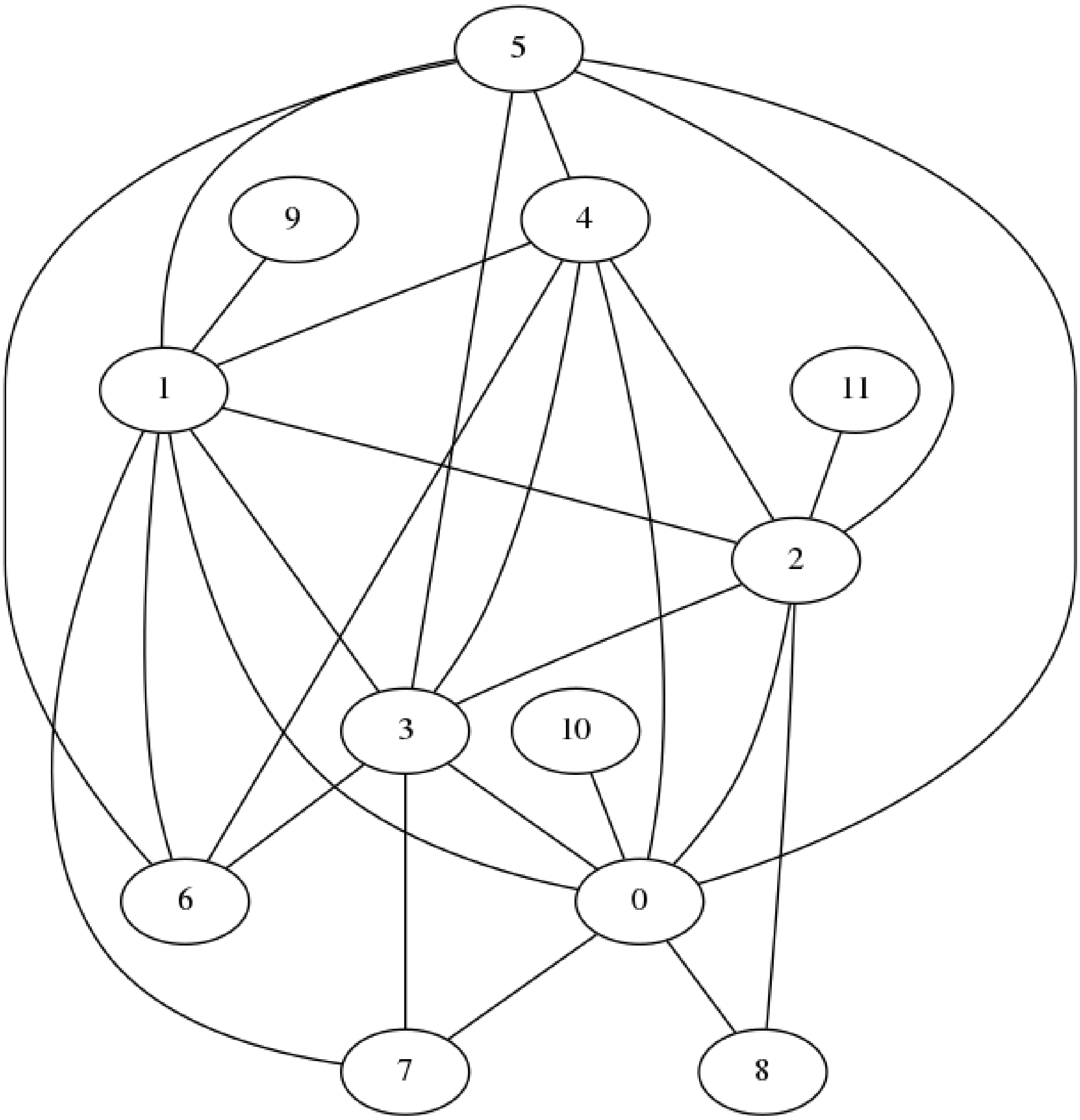
I can solve this using Z3 SMT solver, however, isomorphic graphs are not being weed out... the result is then rendered using GraphViz.

```
#!/usr/bin/env python3

# "The degree sequence problem is the problem of finding some or all graphs with
# the degree sequence being a given non-increasing sequence of positive integers."
# ( https://en.wikipedia.org/wiki/Degree_(graph_theory) )

from z3 import *
import subprocess
```

[5] https://www.amazon.com/Digital-Logic-Verilog-Interview-Questions/dp/1512021466

```
BV_WIDTH=8

# from https://en.wikipedia.org/wiki/Degree_(graph_theory)
#seq=[3, 2, 2, 2, 2, 1, 1, 1]

# from "Pearls in Graph Theory":
#seq=[6,5,5,4,3,3,2,2,2]
#seq=[6,6,6,6,4,3,3,0] # not graphical
#seq=[3,2,1,1,1,1,1]

seq=[8,8,7,7,6,6,4,3,2,1,1,1] # https://math.stackexchange.com/questions/1074651/check-if-sequence-is-graphic-8-8-7-7-6-6-4-3-2-1-1-1

#seq=[1,1]
#seq=[2,2,2]

vertices=len(seq)

if (sum(seq) & 1) == 1:
    print ("not a graphical sequence")
    exit(0)

edges=int(sum(seq)/2)
print ("edges=", edges)

# for each edge, edges_begin[] and edges_end[] pair defines a vertex numbers, which they connect:
edges_begin=[BitVec('edges_begin_%d' % i, BV_WIDTH) for i in range(edges)]
edges_end=[BitVec('edges_end_%d' % i, BV_WIDTH) for i in range(edges)]

# how many times an element encountered in array[]?
def count_elements (array, e):
    rt=[]
    for a in array:
        rt.append(If(a==e, 1, 0))
    return Sum(rt)

s=Solver()

for v in range(vertices):
    #print "vertex %d must be present %d times" % (v, seq[v])
    s.add(count_elements(edges_begin+edges_end, v)==seq[v])

for i in range(edges):
    # no loops must be present
    s.add(edges_begin[i]!=edges_end[i])

    # this is not a multiple graph
    # probably, this is hackish... we say here that each pair of elements (edges_begin[].edges_end[])
    # (where dot is concatenation operation) must not repeat in the arrays, nor in a swapped way
    # this is why edges_[] variables has BitVec type...
    # this can be implemented in other way: a value of edges_begin[]*100+edges_end[] must not appear twice...
    for j in range(edges):
        if i==j:
            continue
        s.add(Concat(edges_begin[i], edges_end[i]) != Concat(edges_begin[j], edges_end[j]))
        s.add(Concat(edges_begin[i], edges_end[i]) != Concat(edges_end[j],
           edges_begin[j]))

gv_no=0

def print_model(m):
    global gv_no
    gv_no=gv_no+1

    print ("edges_begin/edges_end:")
    for i in range(edges):
        print ("%d - %d" % (m[edges_begin[i]].as_long(), m[edges_end[i]].as_long()))

    f=open(str(gv_no)+".gv", "w")
    f.write("graph G {\n")
    for i in range(edges):
        f.write ("\t%d -- %d;\n" % (m[edges_begin[i]].as_long(), m[edges_end[i]].
           as_long()))
    f.write("}\n")
    f.close()
    # run graphviz:
    cmd='dot -Tpng '+str(gv_no)+'.gv -o '+str(gv_no)+'.png'
    print ("running", cmd)
    os.system(cmd)

# enumerate all possible solutions:
results=[]
#while True:
for i in range(10): # 10 results
    if s.check() == sat:
        m = s.model()
        print_model(m)
        #exit(0)
        results.append(m)
        block = []
        for d in m:
            c=d()
            block.append(c != m[d])
        s.add(Or(block))
    else:
        print ("results total=", len(results))
        if len(results)==0:
            print ("not a graphical sequence")
        break
```

For the [8,8,7,7,6,6,4,3,2,1,1,1] sequence I've copypasted from someone's homework...



Exercise

... from the "Pearls in Graph Theory" book:

> 1.1.1. Seven students go on vacations. They decide that each will send a postcard to three of the others. Is it possible that every student receives postcards from precisely the three to whom he sent postcards?

No, it's not possible, because 7*3 is a odd number. However, if you reduce 7 students to 6, this is solvable, the sequence is [3,3,3,3,3,3].

Now the graph of mutual exchanging of postcards between 6 students:

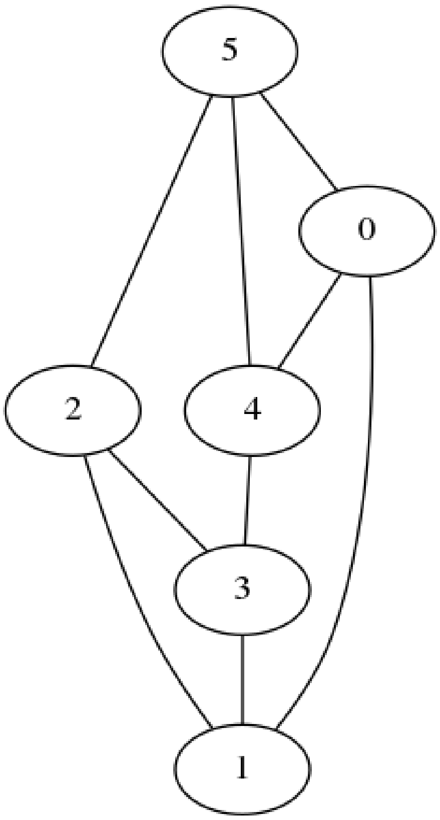


Figure 15.26: 6 students

Chapter 16

Toy decompiler

16.1 Introduction

A modern-day compiler is a product of hundreds of developer/year. At the same time, toy compiler can be an exercise for a student for a week (or even weekend).

Likewise, commercial decompiler like Hex-Rays can be extremely complex, while toy decompiler like this one, can be easy to understand and remake.

The following decompiler written in Python, supports only short basic blocks, with no jumps. Memory is also not supported.

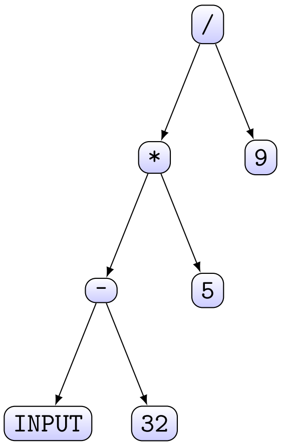
16.2 Data structure

Our toy decompiler will use just one single data structure, representing expression tree.

Many programming textbooks has an example of conversion from Fahrenheit temperature to Celsius, using the following formula:

$$celsius = (fahrenheit - 32) \cdot \frac{5}{9}$$

This expression can be represented as a tree:



How to store it in memory? We see here 3 types of nodes: 1) numbers (or values); 2) arithmetical operations; 3) symbols (like "INPUT").

Many developers with OOP[1] in their mind will create some kind of class. Other developer maybe will use "variant type".

I'll use simplest possible way of representing this structure: a Python tuple. First element of tuple can be a string: either "EXPR_OP" for operation, "EXPR_SYMBOL" for symbol or "EXPR_VALUE" for value. In case of symbol or value, it follows the string. In case of operation, the string followed by another tuples.

Node type and operation type are stored as plain strings—to make debugging output easier to read.

There are *constructors* in our code, in OOP sense:

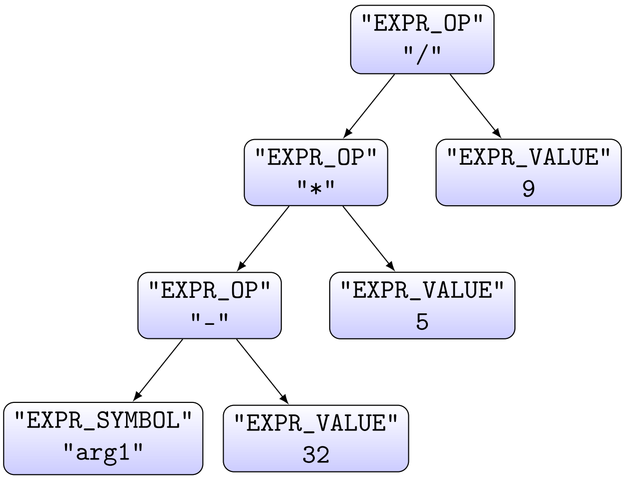
[1] Object-oriented programming

```
def create_val_expr (val):
    return ("EXPR_VALUE", val)

def create_symbol_expr (val):
    return ("EXPR_SYMBOL", val)

def create_binary_expr (op, op1, op2):
    return ("EXPR_OP", op, op1, op2)
```

There are also *accessors*:

```
def get_expr_type(e):
    return e[0]

def get_symbol (e):
    assert get_expr_type(e)=="EXPR_SYMBOL"
    return e[1]

def get_val (e):
    assert get_expr_type(e)=="EXPR_VALUE"
    return e[1]

def is_expr_op(e):
    return get_expr_type(e)=="EXPR_OP"

def get_op (e):
    assert is_expr_op(e)
    return e[1]

def get_op1 (e):
    assert is_expr_op(e)
    return e[2]

def get_op2 (e):
    assert is_expr_op(e)
    return e[3]
```

The temperature conversion expression we just saw will be represented as:



...or as Python expression:

```
('EXPR_OP', '/',
        ('EXPR_OP', '*',
        ('EXPR_OP', '-', ('EXPR_SYMBOL', 'arg1'), ('EXPR_VALUE', 32)),
        ('EXPR_VALUE', 5)),
('EXPR_VALUE', 9))
```

In fact, this is AST[2] in its simplest form. ASTs are used heavily in compilers.

16.3 Simple examples

Let's start with simplest example:

```
        mov     rax, rdi
        imul    rax, rsi
```

At start, these symbols are assigned to registers: RAX=initial_RAX, RBX=initial_RBX, RDI=arg1, RSI=arg2, RDX=arg3, RCX=arg4.

When we handle MOV instruction, we just copy expression from RDI to RAX. When we handle IMUL instruction, we create a new expression, adding together expressions from RAX and RSI and putting result into RAX again.

I can feed this to decompiler and we will see how register's state is changed through processing:

```
python td.py --show-registers --python-expr tests/mul.s

...

line=[mov       rax, rdi]
rcx=('EXPR_SYMBOL', 'arg4')
rsi=('EXPR_SYMBOL', 'arg2')
rbx=('EXPR_SYMBOL', 'initial_RBX')
rdx=('EXPR_SYMBOL', 'arg3')
rdi=('EXPR_SYMBOL', 'arg1')
rax=('EXPR_SYMBOL', 'arg1')

line=[imul      rax, rsi]
rcx=('EXPR_SYMBOL', 'arg4')
rsi=('EXPR_SYMBOL', 'arg2')
rbx=('EXPR_SYMBOL', 'initial_RBX')
rdx=('EXPR_SYMBOL', 'arg3')
rdi=('EXPR_SYMBOL', 'arg1')
rax=('EXPR_OP', '*', ('EXPR_SYMBOL', 'arg1'), ('EXPR_SYMBOL', 'arg2'))

...

result=('EXPR_OP', '*', ('EXPR_SYMBOL', 'arg1'), ('EXPR_SYMBOL', 'arg2'))
```

IMUL instruction is mapped to "*" string, and then new expression is constructed in `handle_binary_op()`, which puts result into RAX.

In this output, the data structures are dumped using Python `str()` function, which does mostly the same, as `print()`.

Output is bulky, and we can turn off Python expressions output, and see how this internal data structure can be rendered neatly using our internal `expr_to_string()` function:

```
python td.py --show-registers tests/mul.s

...

line=[mov       rax, rdi]
rcx=arg4
rsi=arg2
rbx=initial_RBX
rdx=arg3
rdi=arg1
rax=arg1

line=[imul      rax, rsi]
rcx=arg4
rsi=arg2
```

[2] Abstract syntax tree

```
rbx=initial_RBX
rdx=arg3
rdi=arg1
rax=(arg1 * arg2)

...

result=(arg1 * arg2)
```

Slightly advanced example:

```
        imul    rdi, rsi
        lea     rax, [rdi+rdx]
```

LEA instruction is treated just as ADD.

```
python td.py --show-registers --python-expr tests/mul_add.s

...

line=[imul      rdi, rsi]
rcx=('EXPR_SYMBOL', 'arg4')
rsi=('EXPR_SYMBOL', 'arg2')
rbx=('EXPR_SYMBOL', 'initial_RBX')
rdx=('EXPR_SYMBOL', 'arg3')
rdi=('EXPR_OP', '*', ('EXPR_SYMBOL', 'arg1'), ('EXPR_SYMBOL', 'arg2'))
rax=('EXPR_SYMBOL', 'initial_RAX')

line=[lea       rax, [rdi+rdx]]
rcx=('EXPR_SYMBOL', 'arg4')
rsi=('EXPR_SYMBOL', 'arg2')
rbx=('EXPR_SYMBOL', 'initial_RBX')
rdx=('EXPR_SYMBOL', 'arg3')
rdi=('EXPR_OP', '*', ('EXPR_SYMBOL', 'arg1'), ('EXPR_SYMBOL', 'arg2'))
rax=('EXPR_OP', '+', ('EXPR_OP', '*', ('EXPR_SYMBOL', 'arg1'), ('EXPR_SYMBOL', 'arg2
    ')), ('EXPR_SYMBOL', 'arg3'))

...

result=('EXPR_OP', '+', ('EXPR_OP', '*', ('EXPR_SYMBOL', 'arg1'), ('EXPR_SYMBOL', '
    arg2')), ('EXPR_SYMBOL', 'arg3'))
```

And again, let's see this expression dumped neatly:

```
python td.py --show-registers tests/mul_add.s

...

result=((arg1 * arg2) + arg3)
```

Now another example, where we use 2 input arguments:

```
        imul    rdi, rdi, 1234
        imul    rsi, rsi, 5678
        lea     rax, [rdi+rsi]
```

```
python td.py --show-registers --python-expr tests/mul_add3.s

...

line=[imul      rdi, rdi, 1234]
rcx=('EXPR_SYMBOL', 'arg4')
rsi=('EXPR_SYMBOL', 'arg2')
rbx=('EXPR_SYMBOL', 'initial_RBX')
```

```
rdx=('EXPR_SYMBOL', 'arg3')
rdi=('EXPR_OP', '*', ('EXPR_SYMBOL', 'arg1'), ('EXPR_VALUE', 1234))
rax=('EXPR_SYMBOL', 'initial_RAX')

line=[imul      rsi, rsi, 5678]
rcx=('EXPR_SYMBOL', 'arg4')
rsi=('EXPR_OP', '*', ('EXPR_SYMBOL', 'arg2'), ('EXPR_VALUE', 5678))
rbx=('EXPR_SYMBOL', 'initial_RBX')
rdx=('EXPR_SYMBOL', 'arg3')
rdi=('EXPR_OP', '*', ('EXPR_SYMBOL', 'arg1'), ('EXPR_VALUE', 1234))
rax=('EXPR_SYMBOL', 'initial_RAX')

line=[lea       rax, [rdi+rsi]]
rcx=('EXPR_SYMBOL', 'arg4')
rsi=('EXPR_OP', '*', ('EXPR_SYMBOL', 'arg2'), ('EXPR_VALUE', 5678))
rbx=('EXPR_SYMBOL', 'initial_RBX')
rdx=('EXPR_SYMBOL', 'arg3')
rdi=('EXPR_OP', '*', ('EXPR_SYMBOL', 'arg1'), ('EXPR_VALUE', 1234))
rax=('EXPR_OP', '+', ('EXPR_OP', '*', ('EXPR_SYMBOL', 'arg1'), ('EXPR_VALUE', 1234)),
     ('EXPR_OP', '*', ('EXPR_SYMBOL', 'arg2'), ('EXPR_VALUE', 5678)))

...

result=('EXPR_OP', '+', ('EXPR_OP', '*', ('EXPR_SYMBOL', 'arg1'), ('EXPR_VALUE',
    1234)), ('EXPR_OP', '*', ('EXPR_SYMBOL', 'arg2'), ('EXPR_VALUE', 5678)))
```

...and now neat output:

```
python td.py --show-registers tests/mul_add3.s

...

result=((arg1 * 1234) + (arg2 * 5678))
```

Now conversion program:

```
        mov     rax, rdi
        sub     rax, 32
        imul    rax, 5
        mov     rbx, 9
        idiv    rbx
```

You can see, how register's state is changed over execution (or parsing).

Raw:

```
python td.py --show-registers --python-expr tests/fahr_to_celsius.s

...

line=[mov       rax, rdi]
rcx=('EXPR_SYMBOL', 'arg4')
rsi=('EXPR_SYMBOL', 'arg2')
rbx=('EXPR_SYMBOL', 'initial_RBX')
rdx=('EXPR_SYMBOL', 'arg3')
rdi=('EXPR_SYMBOL', 'arg1')
rax=('EXPR_SYMBOL', 'arg1')

line=[sub       rax, 32]
rcx=('EXPR_SYMBOL', 'arg4')
rsi=('EXPR_SYMBOL', 'arg2')
rbx=('EXPR_SYMBOL', 'initial_RBX')
rdx=('EXPR_SYMBOL', 'arg3')
rdi=('EXPR_SYMBOL', 'arg1')
```

```
rax=('EXPR_OP', '-', ('EXPR_SYMBOL', 'arg1'), ('EXPR_VALUE', 32))

line=[imul      rax, 5]
rcx=('EXPR_SYMBOL', 'arg4')
rsi=('EXPR_SYMBOL', 'arg2')
rbx=('EXPR_SYMBOL', 'initial_RBX')
rdx=('EXPR_SYMBOL', 'arg3')
rdi=('EXPR_SYMBOL', 'arg1')
rax=('EXPR_OP', '*', ('EXPR_OP', '-', ('EXPR_SYMBOL', 'arg1'), ('EXPR_VALUE', 32)),
   ('EXPR_VALUE', 5))

line=[mov       rbx, 9]
rcx=('EXPR_SYMBOL', 'arg4')
rsi=('EXPR_SYMBOL', 'arg2')
rbx=('EXPR_VALUE', 9)
rdx=('EXPR_SYMBOL', 'arg3')
rdi=('EXPR_SYMBOL', 'arg1')
rax=('EXPR_OP', '*', ('EXPR_OP', '-', ('EXPR_SYMBOL', 'arg1'), ('EXPR_VALUE', 32)),
   ('EXPR_VALUE', 5))

line=[idiv      rbx]
rcx=('EXPR_SYMBOL', 'arg4')
rsi=('EXPR_SYMBOL', 'arg2')
rbx=('EXPR_VALUE', 9)
rdx=('EXPR_OP', '%', ('EXPR_OP', '*', ('EXPR_OP', '-', ('EXPR_SYMBOL', 'arg1'), ('
   EXPR_VALUE', 32)), ('EXPR_VALUE', 5)), ('EXPR_VALUE', 9))
rdi=('EXPR_SYMBOL', 'arg1')
rax=('EXPR_OP', '/', ('EXPR_OP', '*', ('EXPR_OP', '-', ('EXPR_SYMBOL', 'arg1'), ('
   EXPR_VALUE', 32)), ('EXPR_VALUE', 5)), ('EXPR_VALUE', 9))

...

result=('EXPR_OP', '/', ('EXPR_OP', '*', ('EXPR_OP', '-', ('EXPR_SYMBOL', 'arg1'), ('
   EXPR_VALUE', 32)), ('EXPR_VALUE', 5)), ('EXPR_VALUE', 9))
```

Neat:

```
python td.py --show-registers tests/fahr_to_celsius.s

...

line=[mov       rax, rdi]
rcx=arg4
rsi=arg2
rbx=initial_RBX
rdx=arg3
rdi=arg1
rax=arg1

line=[sub       rax, 32]
rcx=arg4
rsi=arg2
rbx=initial_RBX
rdx=arg3
rdi=arg1
rax=(arg1 - 32)

line=[imul      rax, 5]
rcx=arg4
rsi=arg2
rbx=initial_RBX
rdx=arg3
```

```
rdi=arg1
rax=((arg1 - 32) * 5)

line=[mov       rbx, 9]
rcx=arg4
rsi=arg2
rbx=9
rdx=arg3
rdi=arg1
rax=((arg1 - 32) * 5)

line=[idiv      rbx]
rcx=arg4
rsi=arg2
rbx=9
rdx=(((arg1 - 32) * 5) % 9)
rdi=arg1
rax=(((arg1 - 32) * 5) / 9)

...

result=(((arg1 - 32) * 5) / 9)
```

It is interesting to note that IDIV instruction also calculates reminder of division, and it is placed into RDX register. It's not used, but is available for use.

This is how quotient and remainder are stored in registers:

```
def handle_unary_DIV_IDIV (registers, op1):
    op1_expr=register_or_number_in_string_to_expr (registers, op1)
    current_RAX=registers["rax"]
    registers["rax"]=create_binary_expr ("/", current_RAX, op1_expr)
    registers["rdx"]=create_binary_expr ("%", current_RAX, op1_expr)
```

Now this is `align2grain()` function[3]:

```
        ; uint64_t align2grain (uint64_t i, uint64_t grain)
        ;    return ((i + grain-1) & ~(grain-1));

        ; rdi=i
        ; rsi=grain

        sub     rsi, 1
        add     rdi, rsi
        not     rsi
        and     rdi, rsi
        mov     rax, rdi
```

```
...

line=[sub       rsi, 1]
rcx=arg4
rsi=(arg2 - 1)
rbx=initial_RBX
rdx=arg3
rdi=arg1
rax=initial_RAX

line=[add       rdi, rsi]
rcx=arg4
rsi=(arg2 - 1)
rbx=initial_RBX
```

[3] Taken from https://docs.oracle.com/javase/specs/jvms/se6/html/Compiling.doc.html

```
rdx=arg3
rdi=(arg1 + (arg2 - 1))
rax=initial_RAX

line=[not       rsi]
rcx=arg4
rsi=(~(arg2 - 1))
rbx=initial_RBX
rdx=arg3
rdi=(arg1 + (arg2 - 1))
rax=initial_RAX

line=[and       rdi, rsi]
rcx=arg4
rsi=(~(arg2 - 1))
rbx=initial_RBX
rdx=arg3
rdi=((arg1 + (arg2 - 1)) & (~(arg2 - 1)))
rax=initial_RAX

line=[mov       rax, rdi]
rcx=arg4
rsi=(~(arg2 - 1))
rbx=initial_RBX
rdx=arg3
rdi=((arg1 + (arg2 - 1)) & (~(arg2 - 1)))
rax=((arg1 + (arg2 - 1)) & (~(arg2 - 1)))

...

result=((arg1 + (arg2 - 1)) & (~(arg2 - 1)))
```

16.4 Dealing with compiler optimizations

The following piece of code ...

```
        mov     rax, rdi
        add     rax, rax
```

...will be transformed into *(arg1 + arg1)* expression. It can be reduced to *(arg1 * 2)*. Our toy decompiler can identify patterns like such and rewrite them.

```
# X+X -> X*2
def reduce_ADD1 (expr):
    if is_expr_op(expr) and get_op (expr)=="+" and get_op1 (expr)==get_op2 (expr):
        return dbg_print_reduced_expr ("reduce_ADD1", expr, create_binary_expr ("*",
            get_op1 (expr), create_val_expr (2)))

    return expr # no match
```

This function will just test, if the current node has *EXPR_ OP* type, operation is "+" and both children are equal to each other. By the way, since our data structure is just tuple of tuples, Python can compare them using plain "==" operation. If the testing is finished successfully, current node is then replaced with a new expression: we take one of children, we construct a node of *EXPR_ VALUE* type with "2" number in it, and then we construct a node of *EXPR_ OP* type with "*".

`dbg_print_reduced_expr()` serving solely debugging purposes—it just prints the old and the new (reduced) expressions.

Decompiler is then traverse expression tree recursively in *deep-first search* fashion.

```
def reduce_step (e):
    if is_expr_op (e)==False:
        return e # expr isn't EXPR_OP, nothing to reduce (we don't reduce EXPR_SYMBOL
            and EXPR_VAL)

    if is_unary_op(get_op(e)):
        # recreate expr with reduced operand:
        return reducers(create_unary_expr (get_op(e), reduce_step (get_op1 (e))))
    else:
        # recreate expr with both reduced operands:
        return reducers(create_binary_expr (get_op(e), reduce_step (get_op1 (e)),
           reduce_step (get_op2 (e))))

...


# same as "return ...(reduce_MUL1 (reduce_ADD1 (reduce_ADD2 (... expr))))"
reducers=compose([
        ...
    reduce_ADD1, ...
    ...])

def reduce (e):
    print "going to reduce " + expr_to_string (e)
    new_expr=reduce_step(e)
    if new_expr==e:
        return new_expr # we are done here, expression can't be reduced further
    else:
        return reduce(new_expr) # reduced expr has been changed, so try to reduce it
           again
```

Reduction functions called again and again, as long, as expression changes.

Now we run it:

```
python td.py tests/add1.s

...

going to reduce (arg1 + arg1)
reduction in reduce_ADD1() (arg1 + arg1) -> (arg1 * 2)
going to reduce (arg1 * 2)
result=(arg1 * 2)
```

So far so good, now what if we would try this piece of code?

```
        mov     rax, rdi
        add     rax, rax
        add     rax, rax
        add     rax, rax
```

```
python td.py tests/add2.s

...

working out tests/add2.s
going to reduce (((arg1 + arg1) + (arg1 + arg1)) + ((arg1 + arg1) + (arg1 + arg1)))
reduction in reduce_ADD1() (arg1 + arg1) -> (arg1 * 2)
reduction in reduce_ADD1() (arg1 + arg1) -> (arg1 * 2)
reduction in reduce_ADD1() ((arg1 * 2) + (arg1 * 2)) -> ((arg1 * 2) * 2)
reduction in reduce_ADD1() (arg1 + arg1) -> (arg1 * 2)
reduction in reduce_ADD1() (arg1 + arg1) -> (arg1 * 2)
reduction in reduce_ADD1() ((arg1 * 2) + (arg1 * 2)) -> ((arg1 * 2) * 2)
reduction in reduce_ADD1() (((arg1 * 2) * 2) + ((arg1 * 2) * 2)) -> (((arg1 * 2) * 2)
    * 2)
```

```
going to reduce (((arg1 * 2) * 2) * 2)
result=(((arg1 * 2) * 2) * 2)
```

This is correct, but too verbose.

We would like to rewrite *(X*n)*m* expression to *X*(n*m)*, where n and m are numbers. We can do this by adding another function like `reduce_ADD1()`, but there is much better option: we can make matcher for tree. You can think about it as regular expression matcher, but over trees.

```
def bind_expr (key):
    return ("EXPR_WILDCARD", key)

def bind_value (key):
    return ("EXPR_WILDCARD_VALUE", key)

def match_EXPR_WILDCARD (expr, pattern):
    return {pattern[1] : expr} # return {key : expr}

def match_EXPR_WILDCARD_VALUE (expr, pattern):
    if get_expr_type (expr)!="EXPR_VALUE":
        return None
    return {pattern[1] : get_val(expr)} # return {key : expr}

def is_commutative (op):
    return op in ["+", "*", "&", "|", "^"]

def match_two_ops (op1_expr, op1_pattern, op2_expr, op2_pattern):
    m1=match (op1_expr, op1_pattern)
    m2=match (op2_expr, op2_pattern)
    if m1==None or m2==None:
        return None # one of match for operands returned False, so we do the same
    # join two dicts from both operands:
    rt={}
    rt.update(m1)
    rt.update(m2)
    return rt

def match_EXPR_OP (expr, pattern):
    if get_expr_type(expr)!=get_expr_type(pattern): # be sure, both EXPR_OP.
        return None
    if get_op (expr)!=get_op (pattern): # be sure, ops type are the same.
        return None

    if (is_unary_op(get_op(expr))):
        # match unary expression.
        return match (get_op1 (expr), get_op1 (pattern))
    else:
        # match binary expression.

        # first try match operands as is.
        m=match_two_ops (get_op1 (expr), get_op1 (pattern), get_op2 (expr), get_op2 (
            pattern))
        if m!=None:
            return m
        # if matching unsuccessful, AND operation is commutative, try also swapped
            operands.
        if is_commutative (get_op (expr))==False:
            return None
        return match_two_ops (get_op1 (expr), get_op2 (pattern), get_op2 (expr),
            get_op1 (pattern))

# returns dict in case of success, or None
def match (expr, pattern):
    t=get_expr_type(pattern)
    if t=="EXPR_WILDCARD":
        return match_EXPR_WILDCARD (expr, pattern)
    elif t=="EXPR_WILDCARD_VALUE":
        return match_EXPR_WILDCARD_VALUE (expr, pattern)
    elif t=="EXPR_SYMBOL":
        if expr==pattern:
            return {}
        else:
            return None
    elif t=="EXPR_VALUE":
        if expr==pattern:
            return {}
        else:
            return None
    elif t=="EXPR_OP":
        return match_EXPR_OP (expr, pattern)
    else:
        raise AssertionError
```

Now how we will use it:

```
# (X*A)*B -> X*(A*B)
def reduce_MUL1 (expr):
    m=match (expr, create_binary_expr ("*", (create_binary_expr ("*", bind_expr ("X")
        , bind_value ("A"))), bind_value ("B")))
    if m==None:
        return expr # no match

    return dbg_print_reduced_expr ("reduce_MUL1", expr, create_binary_expr ("*",
        m["X"], # new op1
        create_val_expr (m["A"] * m["B"]))) # new op2
```

We take input expression, and we also construct pattern to be matched. Matcher works recursively over both expressions synchronously. Pattern is also expression, but can use two additional node types: *EXPR_ WILDCARD* and *EXPR_ WILDCARD_ VALUE*. These nodes are supplied with keys (stored as strings). When matcher encounters *EXPR_ WILDCARD* in pattern, it just stashes current expression and will return it. If matcher encounters *EXPR_ WILDCARD_ VALUE*, it does the same, but only in case the current node has *EXPR_ VALUE* type.

`bind_expr()` and `bind_value()` are functions which create nodes with the types we have seen.

All this means, `reduce_MUL1()` function will search for the expression in form *(X*A)*B*, where *A* and *B* are numbers. In other cases, matcher will return input expression untouched, so these reducing function can be chained.

Now when `reduce_MUL1()` encounters (sub)expression we are interesting in, it will return dictionary with keys and expressions. Let's add `print m` call somewhere before return and rerun:

```
python td.py tests/add2.s

...

going to reduce (((arg1 + arg1) + (arg1 + arg1)) + ((arg1 + arg1) + (arg1 + arg1)))
reduction in reduce_ADD1() (arg1 + arg1) -> (arg1 * 2)
reduction in reduce_ADD1() (arg1 + arg1) -> (arg1 * 2)
reduction in reduce_ADD1() ((arg1 * 2) + (arg1 * 2)) -> ((arg1 * 2) * 2)
{'A': 2, 'X': ('EXPR_SYMBOL', 'arg1'), 'B': 2}
reduction in reduce_MUL1() ((arg1 * 2) * 2) -> (arg1 * 4)
reduction in reduce_ADD1() (arg1 + arg1) -> (arg1 * 2)
reduction in reduce_ADD1() (arg1 + arg1) -> (arg1 * 2)
reduction in reduce_ADD1() ((arg1 * 2) + (arg1 * 2)) -> ((arg1 * 2) * 2)
{'A': 2, 'X': ('EXPR_SYMBOL', 'arg1'), 'B': 2}
reduction in reduce_MUL1() ((arg1 * 2) * 2) -> (arg1 * 4)
reduction in reduce_ADD1() ((arg1 * 4) + (arg1 * 4)) -> ((arg1 * 4) * 2)
{'A': 4, 'X': ('EXPR_SYMBOL', 'arg1'), 'B': 2}
reduction in reduce_MUL1() ((arg1 * 4) * 2) -> (arg1 * 8)
going to reduce (arg1 * 8)
```

```
...
result=(arg1 * 8)
```

The dictionary has keys we supplied plus expressions matcher found. We then can use them to create new expression and return it. Numbers are just summed while forming second operand to "*" operation.

Now a real-world optimization technique—optimizing GCC replaced multiplication by 31 by shifting and subtraction operations:

```
        mov     rax, rdi
        sal     rax, 5
        sub     rax, rdi
```

Without reduction functions, our decompiler will translate this into *((arg1 « 5) - arg1).* We can replace shifting left by multiplication:

```
# X<<n -> X*(2^n)
def reduce_SHL1 (expr):
    m=match (expr, create_binary_expr ("<<", bind_expr ("X"), bind_value ("Y")))
    if m==None:
        return expr # no match

    return dbg_print_reduced_expr ("reduce_SHL1", expr, create_binary_expr ("*", m["X
       "], create_val_expr (1<<m["Y"])))
```

Now we getting *((arg1 * 32) - arg1).* We can add another reduction function:

```
# (X*n)-X -> X*(n-1)
def reduce_SUB3 (expr):
    m=match (expr, create_binary_expr ("-",
        create_binary_expr ("*", bind_expr("X1"), bind_value ("N")),
        bind_expr("X2")))

    if m!=None and match (m["X1"], m["X2"])!=None:
        return dbg_print_reduced_expr ("reduce_SUB3", expr, create_binary_expr ("*",
           m["X1"], create_val_expr (m["N"]-1)))
    else:
        return expr # no match
```

Matcher will return two X's, and we must be assured that they are equal. In fact, in previous versions of this toy decompiler, I did comparison with plain "==", and it worked. But we can reuse `match()` function for the same purpose, because it will process commutative operations better. For example, if X1 is "Q+1" and X2 is "1+Q", expressions are equal, but plain "==" will not work. On the other side, `match()` function, when encounter "+" operation (or another commutative operation), and it fails with comparison, it will also try swapped operand and will try to compare again.

However, to understand it easier, for a moment, you can imagine there is "==" instead of the second `match()`.

Anyway, here is what we've got:

```
working out tests/mul31_GCC.s
going to reduce ((arg1 << 5) - arg1)
reduction in reduce_SHL1() (arg1 << 5) -> (arg1 * 32)
reduction in reduce_SUB3() ((arg1 * 32) - arg1) -> (arg1 * 31)
going to reduce (arg1 * 31)
...
result=(arg1 * 31)
```

Another optimization technique is often seen in ARM thumb code: AND-ing a value with a value like 0xFFFFFFF0, is implemented using shifts:

```
        mov rax, rdi
        shr rax, 4
        shl rax, 4
```

This code is quite common in ARM thumb code, because it's a headache to encode 32-bit constants using couple of 16-bit thumb instructions, while single 16-bit instruction can shift by 4 bits left or right.

Also, the expression *(x»4)«4* can be jokingly called as "twitching operator": I've heard the "--i++" expression was called like this in Russian-speaking social networks, it was some kind of meme ("operator podergivaniya").

Anyway, these reduction functions will be used:

```
# X>>n -> X / (2^n)
...
def reduce_SHR2 (expr):
    m=match(expr, create_binary_expr(">>", bind_expr("X"), bind_value("Y")))
    if m==None or m["Y"]>=64:
        return expr # no match

    return dbg_print_reduced_expr ("reduce_SHR2", expr, create_binary_expr ("/", m["X
       "],
        create_val_expr (1<<m["Y"])))

...

# X<<n -> X*(2^n)
def reduce_SHL1 (expr):
    m=match (expr, create_binary_expr ("<<", bind_expr ("X"), bind_value ("Y")))
    if m==None:
        return expr # no match

    return dbg_print_reduced_expr ("reduce_SHL1", expr, create_binary_expr ("*", m["X
       "], create_val_expr (1<<m["Y"])))

...

# FIXME: slow
# returns True if n=2^x or popcnt(n)=1
def is_2n(n):
    return bin(n).count("1")==1


# AND operation using DIV/MUL or SHL/SHR
# (X / (2^n)) * (2^n) -> X&(~((2^n)-1))
def reduce_MUL2 (expr):
    m=match(expr, create_binary_expr ("*", create_binary_expr ("/", bind_expr("X"),
       bind_value("N1")), bind_value("N2")))
    if m==None or m["N1"]!=m["N2"] or is_2n(m["N1"])==False: # short-circuit
       expression
        return expr # no match

    return dbg_print_reduced_expr("reduce_MUL2", expr, create_binary_expr ("&", m["X
       "],
        create_val_expr(~(m["N1"]-1)&0xffffffffffffffff)))
```

Now the result:

```
working out tests/AND_by_shifts2.s
going to reduce ((arg1 >> 4) << 4)
reduction in reduce_SHR2() (arg1 >> 4) -> (arg1 / 16)
reduction in reduce_SHL1() ((arg1 / 16) << 4) -> ((arg1 / 16) * 16)
reduction in reduce_MUL2() ((arg1 / 16) * 16) -> (arg1 & 0xfffffffffffffff0)
going to reduce (arg1 & 0xfffffffffffffff0)
...
result=(arg1 & 0xfffffffffffffff0)
```

16.4.1 Division using multiplication

Division is often replaced by multiplication for performance reasons.

From school-level arithmetic, we can remember that division by 3 can be replaced by multiplication by $\frac{1}{3}$. In fact, sometimes compilers do so for floating-point arithmetic, for example, FDIV instruction in x86 code can be replaced by FMUL. At least MSVC 6.0 will replace division by 3 by multiplication by $\frac{1}{3}$ and sometimes it's hard to be sure, what operation was in original source code.

But when we operate over integer values and CPU registers, we can't use fractions. However, we can rework fraction:

$$result = \frac{x}{3} = x \cdot \frac{1}{3} = x \cdot \frac{1 \cdot MagicNumber}{3 \cdot MagicNumber}$$

Given the fact that division by 2^n is very fast, we now should find that $MagicNumber$, for which the following equation will be true: $2^n = 3 \cdot MagicNumber$.

This code performing division by 10:

```
        mov     rax, rdi
        movabs  rdx, 0cccccccccccccccdh
        mul     rdx
        shr     rdx, 3
        mov     rax, rdx
```

Division by 2^{64} is somewhat hidden: lower 64-bit of product in RAX is not used (dropped), only higher 64-bit of product (in RDX) is used and then shifted by additional 3 bits.

RDX register is set during processing of MUL/IMUL like this:

```
def handle_unary_MUL_IMUL (registers, op1):
    op1_expr=register_or_number_in_string_to_expr (registers, op1)
    result=create_binary_expr ("*", registers["rax"], op1_expr)
    registers["rax"]=result
    registers["rdx"]=create_binary_expr (">>", result, create_val_expr(64))
```

In other words, the assembly code we have just seen multiplication operations by $\frac{0cccccccccccccccdh}{2^{64+3}}$, or divides by $\frac{2^{64+3}}{0cccccccccccccccdh}$. To find divisor we just have to divide numerator by denominator.

```
# n = magic number
# m = shifting coefficient
# return = 1 / (n / 2^m) = 2^m / n
def get_divisor (n, m):
    return (2**float(m))/float(n)

# (X*n)>>m, where m>=64 -> X/...
def reduce_div_by_MUL (expr):
    m=match (expr, create_binary_expr(">>", create_binary_expr ("*", bind_expr("X"),
        bind_value("N")), bind_value("M")))
    if m==None:
        return expr # no match

    divisor=get_divisor(m["N"], m["M"])
    return dbg_print_reduced_expr ("reduce_div_by_MUL", expr, create_binary_expr
        ("/", m["X"], create_val_expr (int(divisor))))
```

This works, but we have a problem: this rule takes *(arg1 * 0xcccccccccccccccd) » 64* expression first and finds divisor to be equal to 1.25. This is correct: result is shifted by 3 bits after (or divided by 8), and $1.25 \cdot 8 = 10$. But our toy decompiler doesn't support real numbers.

We can solve this problem in the following way: if divisor has fractional part, we postpone reducing, with a hope, that two subsequent right shift operations will be reduced into single one:

```
# (X*n)>>m, where m>=64 -> X/...
def reduce_div_by_MUL (expr):
    m=match (expr, create_binary_expr(">>", create_binary_expr ("*", bind_expr("X"),
        bind_value("N")), bind_value("M")))
    if m==None:
        return expr # no match

    divisor=get_divisor(m["N"], m["M"])
    if math.floor(divisor)==divisor:
        return dbg_print_reduced_expr ("reduce_div_by_MUL", expr, create_binary_expr
            ("/", m["X"], create_val_expr (int(divisor))))
    else:
        print "reduce_div_by_MUL(): postponing reduction, because divisor=", divisor
        return expr
```

That works:

```
working out tests/div_by_mult10_unsigned.s
going to reduce (((arg1 * 0xcccccccccccccccd) >> 64) >> 3)
reduce_div_by_MUL(): postponing reduction, because divisor= 1.25
reduction in reduce_SHR1() (((arg1 * 0xcccccccccccccccd) >> 64) >> 3) -> ((arg1 * 0
   xcccccccccccccccd) >> 67)
going to reduce ((arg1 * 0xcccccccccccccccd) >> 67)
reduction in reduce_div_by_MUL() ((arg1 * 0xcccccccccccccccd) >> 67) -> (arg1 / 10)
going to reduce (arg1 / 10)
result=(arg1 / 10)
```

I don't know if this is best solution. In early version of this decompiler, it processed input expression in two passes: first pass for everything except division by multiplication, and the second pass for the latter. I don't know which way is better. Or maybe we could support real numbers in expressions?

Couple of words about better understanding division by multiplication. Many people miss "hidden" division by 2^{32} or 2^{64}, when lower 32-bit part (or 64-bit part) of product is not used (or just dropped). Also, there is misconception that modulo inverse is used here. This is close, but not the same thing. Extended Euclidean algorithm is usually used to find *magic coefficient*, but in fact, this algorithm is rather used to solve the equation. You can solve it using any other method. Also, needless to mention, the equation is unsolvable for some divisors, because this is diophantine equation (i.e., equation allowing result to be only integer), since we work on integer CPU registers, after all.

16.5 Obfuscation/deobfuscation

Despite simplicity of our decompiler, we can see how to deobfuscate (or optimize) using several simple tricks.

For example, this piece of code does nothing:

```
        mov rax, rdi
        xor rax, 12345678h
        xor rax, 0deadbeefh
        xor rax, 12345678h
        xor rax, 0deadbeefh
```

We would need these rules to tame it:

```
# (X^n)^m -> X^(n^m)
def reduce_XOR4 (expr):
    m=match(expr,
        create_binary_expr("^",
            create_binary_expr ("^", bind_expr("X"), bind_value("N")),
                bind_value("M")))

    if m!=None:
        return dbg_print_reduced_expr ("reduce_XOR4", expr, create_binary_expr ("^",
           m["X"],
            create_val_expr (m["N"]^m["M"])))
    else:
        return expr # no match

...

# X op 0 -> X, where op is ADD, OR, XOR, SUB
def reduce_op_0 (expr):
    # try each:
    for op in ["+", "|", "^", "-"]:
        m=match(expr, create_binary_expr(op, bind_expr("X"), create_val_expr (0)))
        if m!=None:
            return dbg_print_reduced_expr ("reduce_op_0", expr, m["X"])

    # default:
```

```
    return expr # no match
```

```
working out tests/t9_obf.s
going to reduce ((((arg1 ^ 0x12345678) ^ 0xdeadbeef) ^ 0x12345678) ^ 0xdeadbeef)
reduction in reduce_XOR4() ((arg1 ^ 0x12345678) ^ 0xdeadbeef) -> (arg1 ^ 0xcc99e897)
reduction in reduce_XOR4() ((arg1 ^ 0xcc99e897) ^ 0x12345678) -> (arg1 ^ 0xdeadbeef)
reduction in reduce_XOR4() ((arg1 ^ 0xdeadbeef) ^ 0xdeadbeef) -> (arg1 ^ 0x0)
going to reduce (arg1 ^ 0x0)
reduction in reduce_op_0() (arg1 ^ 0x0) -> arg1
going to reduce arg1
result=arg1
```

This piece of code can be deobfuscated (or optimized) as well:

```
; toggle last bit

        mov rax, rdi
        mov rbx, rax
        mov rcx, rbx
        mov rsi, rcx
        xor rsi, 12345678h
        xor rsi, 12345679h
        mov rax, rsi
```

```
working out tests/t7_obf.s
going to reduce ((arg1 ^ 0x12345678) ^ 0x12345679)
reduction in reduce_XOR4() ((arg1 ^ 0x12345678) ^ 0x12345679) -> (arg1 ^ 1)
going to reduce (arg1 ^ 1)
result=(arg1 ^ 1)
```

I also used *aha!*[4] superoptimizer to find weird piece of code which does nothing.

Aha! is so called superoptimizer, it tries various piece of codes in brute-force manner, in attempt to find shortest possible alternative for some mathematical operation. While sane compiler developers use superoptimizers for this task, I tried it in opposite way, to find oddest pieces of code for some simple operations, including NOP operation. In past, I've used it to find weird alternative to XOR operation (4.1).

So here is what *aha!* can find for NOP:

```
; do nothing (as found by aha)

        mov rax, rdi
        and rax, rax
        or rax, rax
```

```
# X & X -> X
def reduce_AND3 (expr):
    m=match (expr, create_binary_expr ("&", bind_expr ("X1"), bind_expr ("X2")))
    if m!=None and match (m["X1"], m["X2"])!=None:
        return dbg_print_reduced_expr("reduce_AND3", expr, m["X1"])
    else:
        return expr # no match

...

# X | X -> X
def reduce_OR1 (expr):
    m=match (expr, create_binary_expr ("|", bind_expr ("X1"), bind_expr ("X2")))
    if m!=None and match (m["X1"], m["X2"])!=None:
        return dbg_print_reduced_expr("reduce_OR1", expr, m["X1"])
    else:
        return expr # no match
```

[4] http://www.hackersdelight.org/aha/aha.pdf

```
working out tests/t11_obf.s
going to reduce ((arg1 & arg1) | (arg1 & arg1))
reduction in reduce_AND3() (arg1 & arg1) -> arg1
reduction in reduce_AND3() (arg1 & arg1) -> arg1
reduction in reduce_OR1() (arg1 | arg1) -> arg1
going to reduce arg1
result=arg1
```

This is weirder:

```
; do nothing (as found by aha)

;Found a 5-operation program:
;   neg   r1,rx
;   neg   r2,rx
;   neg   r3,r1
;   or    r4,rx,2
;   and   r5,r4,r3
;   Expr: ((x | 2) & -(-(x)))

        mov rax, rdi
        neg rax
        neg rax
        or rdi, 2
        and rax, rdi
```

Rules added (I used “NEG” string to represent sign change and to be different from subtraction operation, which is just minus (“-”)):

```
# (op(op X)) -> X, where both ops are NEG or NOT
def reduce_double_NEG_or_NOT (expr):
    # try each:
    for op in ["NEG", "~"]:
        m=match (expr, create_unary_expr (op, create_unary_expr (op, bind_expr("X")))
            )
        if m!=None:
            return dbg_print_reduced_expr ("reduce_double_NEG_or_NOT", expr, m["X"])

    # default:
    return expr # no match

...

# X & (X | ...) -> X
def reduce_AND2 (expr):
    m=match (expr, create_binary_expr ("&", create_binary_expr ("|", bind_expr ("X1")
        , bind_expr ("REST")), bind_expr ("X2")))
    if m!=None and match (m["X1"], m["X2"])!=None:
        return dbg_print_reduced_expr("reduce_AND2", expr, m["X1"])
    else:
        return expr # no match
```

```
going to reduce ((-(-arg1)) & (arg1 | 2))
reduction in reduce_double_NEG_or_NOT() (-(-arg1)) -> arg1
reduction in reduce_AND2() (arg1 & (arg1 | 2)) -> arg1
going to reduce arg1
result=arg1
```

I also forced *aha!* to find piece of code which adds 2 with no addition/subtraction operations allowed:

```
; arg1+2, without add/sub allowed, as found by aha:
```

```
;Found a 4-operation program:
;   not   r1,rx
;   neg   r2,r1
;   not   r3,r2
;   neg   r4,r3
;   Expr: -(~(-(~(x))))

        mov     rax, rdi
        not     rax
        neg     rax
        not     rax
        neg     rax
```

Rule:

```
# (- (~X)) -> X+1
def reduce_NEG_NOT (expr):
    m=match (expr, create_unary_expr ("NEG", create_unary_expr ("~", bind_expr("X")))
        )
    if m==None:
        return expr # no match

    return dbg_print_reduced_expr ("reduce_NEG_NOT", expr, create_binary_expr ("+", m
        ["X"],create_val_expr(1)))
```

```
working out tests/add_by_not_neg.s
going to reduce (-(~(-(~arg1))))
reduction in reduce_NEG_NOT() (-(~arg1)) -> (arg1 + 1)
reduction in reduce_NEG_NOT() (-(~(arg1 + 1))) -> ((arg1 + 1) + 1)
reduction in reduce_ADD3() ((arg1 + 1) + 1) -> (arg1 + 2)
going to reduce (arg1 + 2)
result=(arg1 + 2)
```

This is artifact of two's complement system of signed numbers representation. Same can be done for subtraction (just swap NEG and NOT operations).

Now let's add some fake luggage to Fahrenheit-to-Celsius example:

```
        ; celsius = 5 * (fahr-32) / 9
        ; fake luggage:
        mov     rbx, 12345h
        mov     rax, rdi
        sub     rax, 32
        ; fake luggage:
        add     rbx, rax
        imul    rax, 5
        mov     rbx, 9
        idiv    rbx
        ; fake luggage:
        sub     rdx, rax
```

It's not a problem for our decompiler, because the noise is left in RDX register, and not used at all:

```
working out tests/fahr_to_celsius_obf1.s
line=[mov       rbx, 12345h]
rcx=arg4
rsi=arg2
rbx=0x12345
rdx=arg3
rdi=arg1
rax=initial_RAX

line=[mov       rax, rdi]
rcx=arg4
```

```
rsi=arg2
rbx=0x12345
rdx=arg3
rdi=arg1
rax=arg1

line=[sub       rax, 32]
rcx=arg4
rsi=arg2
rbx=0x12345
rdx=arg3
rdi=arg1
rax=(arg1 - 32)

line=[add       rbx, rax]
rcx=arg4
rsi=arg2
rbx=(0x12345 + (arg1 - 32))
rdx=arg3
rdi=arg1
rax=(arg1 - 32)

line=[imul      rax, 5]
rcx=arg4
rsi=arg2
rbx=(0x12345 + (arg1 - 32))
rdx=arg3
rdi=arg1
rax=((arg1 - 32) * 5)

line=[mov       rbx, 9]
rcx=arg4
rsi=arg2
rbx=9
rdx=arg3
rdi=arg1
rax=((arg1 - 32) * 5)

line=[idiv      rbx]
rcx=arg4
rsi=arg2
rbx=9
rdx=(((arg1 - 32) * 5) % 9)
rdi=arg1
rax=(((arg1 - 32) * 5) / 9)

line=[sub       rdx, rax]
rcx=arg4
rsi=arg2
rbx=9
rdx=((((arg1 - 32) * 5) % 9) - (((arg1 - 32) * 5) / 9))
rdi=arg1
rax=(((arg1 - 32) * 5) / 9)

going to reduce (((arg1 - 32) * 5) / 9)
result=(((arg1 - 32) * 5) / 9)
```

We can try to pretend we affect the result with the noise:

```
        ; celsius = 5 * (fahr-32) / 9
        ; fake luggage:
        mov     rbx, 12345h
        mov     rax, rdi
        sub     rax, 32
        ; fake luggage:
        add     rbx, rax
        imul    rax, 5
        mov     rbx, 9
        idiv    rbx
        ; fake luggage:
        sub     rdx, rax
        mov     rcx, rax
        ; OR result with garbage (result of fake luggage):
        or      rcx, rdx
        ; the following instruction shouldn't affect result:
        and     rax, rcx
```

...but in fact, it's all reduced by `reduce_AND2()` function we already saw (16.5):

```
working out tests/fahr_to_celsius_obf2.s
going to reduce ((((arg1 - 32) * 5) / 9) & ((((arg1 - 32) * 5) / 9) | ((((arg1 - 32)
   * 5) % 9) - (((arg1 - 32) * 5) / 9))))
reduction in reduce_AND2() ((((arg1 - 32) * 5) / 9) & ((((arg1 - 32) * 5) / 9) | ((((
   arg1 - 32) * 5) % 9) - (((arg1 - 32) * 5)
/ 9)))) -> (((arg1 - 32) * 5) / 9)
going to reduce (((arg1 - 32) * 5) / 9)
result=(((arg1 - 32) * 5) / 9)
```

We can see that deobfuscation is in fact the same thing as optimization used in compilers. We can try this function in GCC:

```
int f(int a)
{
        return -(~a);
};
```

Optimizing GCC 5.4 (x86) generates this:

```
f:
        mov     eax, DWORD PTR [esp+4]
        add     eax, 1
        ret
```

GCC has its own rewriting rules, some of which are, probably, close to what we use here.

16.6 Tests

Despite simplicity of the decompiler, it's still error-prone. We need to be sure that original expression and reduced one are equivalent to each other.

16.6.1 Evaluating expressions

First of all, we would just evaluate (or *run*, or *execute*) expression with random values as arguments, and then compare results.

Evaluator do arithmetical operations when possible, recursively. When any symbol is encountered, its value (randomly generated before) is taken from a table.

```
un_ops={"NEG":operator.neg,
        "~":operator.invert}

bin_ops={">>":operator.rshift,
        "<<":(lambda x, c: x<<(c&0x3f)), # operator.lshift should be here, but it
           doesn't handle too big counts
        "&":operator.and_,
        "|":operator.or_,
        "^":operator.xor,
        "+":operator.add,
        "-":operator.sub,
        "*":operator.mul,
        "/":operator.div,
        "%":operator.mod}

def eval_expr(e, symbols):
    t=get_expr_type (e)
    if t=="EXPR_SYMBOL":
        return symbols[get_symbol(e)]
    elif t=="EXPR_VALUE":
        return get_val (e)
    elif t=="EXPR_OP":
        if is_unary_op (get_op (e)):
            return un_ops[get_op(e)](eval_expr(get_op1(e), symbols))
        else:
            return bin_ops[get_op(e)](eval_expr(get_op1(e), symbols), eval_expr(
               get_op2(e), symbols))
    else:
        raise AssertionError

def do_selftest(old, new):
    for n in range(100):
        symbols={"arg1":random.getrandbits(64),
                "arg2":random.getrandbits(64),
                "arg3":random.getrandbits(64),
                "arg4":random.getrandbits(64)}
        old_result=eval_expr (old, symbols)&0xffffffffffffffff # signed->unsigned
        new_result=eval_expr (new, symbols)&0xffffffffffffffff # signed->unsigned
        if old_result!=new_result:
            print "self-test failed"
            print "initial expression: "+expr_to_string(old)
            print "reduced expression: "+expr_to_string(new)
            print "initial expression result: ", old_result
            print "reduced expression result: ", new_result
            exit(0)
```

In fact, this is very close to what LISP *EVAL* function does, or even LISP interpreter. However, not all symbols are set. If the expression is using initial values from RAX or RBX (to which symbols "initial_RAX" and "initial_RBX" are assigned, decompiler will stop with exception, because no random values assigned to these registers, and these symbols are absent in *symbols* dictionary.

Using this test, I've suddenly found a bug here (despite simplicity of all these reduction rules). Well, no-one protected from eye strain. Nevertheless, the test has a serious problem: some bugs can be revealed only if one of arguments is 0, or 1, or −1. Maybe there are even more special cases exists.

Mentioned above *aha!* superoptimizer tries at least these values as arguments while testing: 1, 0, -1, 0x80000000, 0x7FFFFFFF, 0x80000001, 0x7FFFFFFE, 0x01234567, 0x89ABCDEF, -2, 2, -3, 3, -64, 64, -5, -31415.

Still, you cannot be sure.

16.6.2 Using Z3 SMT-solver for testing

So here we will try Z3 SMT-solver. SMT-solver can *prove* that two expressions are equivalent to each other.

For example, with the help of *aha!*, I've found another weird piece of code, which does nothing:

```
; do nothing (obfuscation)

;Found a 5-operation program:
;   neg   r1,rx
;   neg   r2,r1
;   sub   r3,r1,3
;   sub   r4,r3,r1
;   sub   r5,r4,r3
;    Expr: (((-(x) - 3) - -(x)) - (-(x) - 3))

        mov rax, rdi
        neg rax
        mov rbx, rax
        ; rbx=-x
        mov rcx, rbx
        sub rcx, 3
        ; rcx=-x-3
        mov rax, rcx
        sub rax, rbx
        ; rax=(-(x) - 3) - -(x)
        sub rax, rcx
```

Using toy decompiler, I've found that this piece is reduced to *arg1* expression:

```
working out tests/t5_obf.s
going to reduce ((((-arg1) - 3) - (-arg1)) - ((-arg1) - 3))
reduction in reduce_SUB2() ((-arg1) - 3) -> (-(arg1 + 3))
reduction in reduce_SUB5() ((-(arg1 + 3)) - (-arg1)) -> ((-(arg1 + 3)) + arg1)
reduction in reduce_SUB2() ((-arg1) - 3) -> (-(arg1 + 3))
reduction in reduce_ADD_SUB() (((-(arg1 + 3)) + arg1) - (-(arg1 + 3))) -> arg1
going to reduce arg1
result=arg1
```

But is it correct? I've added a function which can output expression(s) to SMT-LIB-format, it's as simple as a function which converts expression to string.

And this is SMT-LIB-file for Z3:

```
(assert
    (forall ((arg1 (_ BitVec 64)) (arg2 (_ BitVec 64)) (arg3 (_ BitVec 64)) (arg4 (_
       BitVec 64)))
        (=
            (bvsub (bvsub (bvsub (bvneg arg1) #x0000000000000003) (bvneg arg1)) (
               bvsub (bvneg arg1) #x0000000000000003))
            arg1
        )
    )
)
(check-sat)
```

In plain English terms, what we asking it to be sure, that *forall* four 64-bit arguments, two expressions are equivalent (second is just *arg1*).

The syntax maybe hard to understand, but in fact, this is very close to LISP, and arithmetical operations are named “bvsub”, “bvadd”, etc, because “bv” stands for *bit vector*.

While running, Z3 shows “sat”, meaning “satisfiable”. In other words, Z3 couldn't find counterexample for this expression.

In fact, I can rewrite this expression in the following form: *expr1 != expr2*, and we would ask Z3 to find at least one set of input arguments, for which expressions are not equal to each other:

```
(declare-const arg1 (_ BitVec 64))
(declare-const arg2 (_ BitVec 64))
(declare-const arg3 (_ BitVec 64))
(declare-const arg4 (_ BitVec 64))

(assert
    (not
        (=
            (bvsub (bvsub (bvsub (bvneg arg1) #x0000000000000003) (bvneg arg1)) (
               bvsub (bvneg arg1) #x0000000000000003))
            arg1
        )
    )
)
(check-sat)
```

Z3 says “unsat”, meaning, it couldn’t find any such counterexample. In other words, for all possible input arguments, results of these two expressions are always equal to each other.

Nevertheless, Z3 is not omnipotent. It fails to prove equivalence of the code which performs division by multiplication. First of all, I extended it so both results will have size of 128 bit instead of 64:

```
(declare-const x (_ BitVec 64))
    (assert
        (forall ((x (_ BitVec 64)))
            (=
                ((_ zero_extend 64) (bvudiv x (_ bv17 64)))
                (bvlshr (bvmul ((_ zero_extend 64) x) #
                   x0000000000000000f0f0f0f0f0f0f0f1) (_ bv68 128))
            )
        )
    )
(check-sat)
(get-model)
```

(*bv17* is just 64-bit number 17, etc. “bv” stands for “bit vector”, as opposed to integer value.)

Z3 works too long without any answer, and I had to interrupt it.

As Z3 developers mentioned, such expressions are hard for Z3 so far: https://github.com/Z3Prover/z3/issues/514.

Still, division by multiplication can be tested using previously described brute-force check.

16.7 My other implementations of toy decompiler

When I made attempt to write it in C++, of course, node in expression was represented using class. There is also implementation in pure C[5], node is represented using structure.

Matchers in both C++ and C versions doesn’t return any dictionary, but instead, `bind_value()` functions takes pointer to a variable which will contain value after successful matching. `bind_expr()` takes pointer to a pointer, which will points to the part of expression, again, in case of success. I took this idea from LLVM.

Here are two pieces of code from LLVM source code with couple of reducing rules:

```
// (X >> A) << A -> X
  Value *X;
  if (match(Op0, m_Exact(m_Shr(m_Value(X), m_Specific(Op1)))))
    return X;
```

(lib/Analysis/InstructionSimplify.cpp)

```
// (A | B) | C  and  A | (B | C)                  -> bswap if possible.
  // (A >> B) | (C << D)  and  (A << B) | (B >> C)  -> bswap if possible.
  if (match(Op0, m_Or(m_Value(), m_Value())) ||
      match(Op1, m_Or(m_Value(), m_Value())) ||
      (match(Op0, m_LogicalShift(m_Value(), m_Value())) &&
       match(Op1, m_LogicalShift(m_Value(), m_Value())))) {
    if (Instruction *BSwap = MatchBSwap(I))
      return BSwap;
```

(lib/Transforms/InstCombine/InstCombineAndOrXor.cpp)

As you can see, my matcher tries to mimic LLVM. What I call *reduction* is called *folding* in LLVM. Both terms are popular.

I have also a blog post about LLVM obfuscator, in which LLVM matcher is mentioned: https://yurichev.com/blog/llvm/.

Python version of toy decompiler uses strings in place where enumerate data type is used in C version (like *OP_AND*, *OP_MUL*, etc) and symbols used in Racket version[6] (like *’OP_DIV*, etc). This may be seen as inefficient, nevertheless, thanks to strings interning, only address of strings are compared in Python version, not strings as a whole. So strings in Python can be seen as possible replacement for LISP symbols.

[5] https://smt.st/current_tree/toy_decompiler/files/C

[6] Racket is Scheme (which is, in turn, LISP dialect) dialect. https://smt.st/current_tree/toy_decompiler/files/Racket

16.7.1 Even simpler toy decompiler

Knowledge of LISP makes you understand all these things naturally, without significant effort. But when I had no knowledge of it, but still tried to make a simple toy decompiler, I made it using usual text strings which hold expressions for each registers (and even memory).

So when MOV instruction copies value from one register to another, we just copy string. When arithmetical instruction occurred, we do string concatenation:

```
std::string registers[TOTAL];

...

// all 3 arguments are strings
switch (ins, op1, op2)
{
	...
	case ADD:	registers[op1]="(" + registers[op1] + " + " + registers[op2] + ")";
			break;
	...
	case MUL:	registers[op1]="(" + registers[op1] + " / " + registers[op2] + ")";
			break;
	...
}
```

Now you'll have long expressions for each register, represented as strings. For reducing them, you can use plain simple regular expression matcher.

For example, for the rule `(X*n)+(X*m) -> X*(n+m)`, you can match (sub)string using the following regular expression:
`((.*)*(.*))+((.*)*(.*))` [7]. If the string is matched, you're getting 4 groups (or substrings). You then just compare 1st and 3rd using string comparison function, then you check if the 2nd and 4th are numbers, you convert them to numbers, sum them and you make new string, consisting of 1st group and sum, like this: `(" + X + "*" + (int(n) + int(m)) + ")`.

It was naïve, clumsy, it was source of great embarrassment, but it worked correctly.

16.8 Difference between toy decompiler and commercial-grade one

Perhaps, someone, who currently reading this text, may rush into extending my source code. As an exercise, I would say, that the first step could be support of partial registers: i.e., AL, AX, EAX. This is tricky, but doable.

Another task may be support of FPU[8] x86 instructions (FPU stack modeling isn't a big deal).

The gap between toy decompiler and a commercial decompiler like Hex-Rays is still enormous. Several tricky problems must be solved, at least these:

- C data types: arrays, structures, pointers, etc. This problem is virtually non-existent for JVM[9] (Java, etc) and .NET decompilers, because type information is present in binary files.
- Basic blocks, C/C++ statements. Mike Van Emmerik in his thesis [10] shows how this can be tackled using SSA forms (which are also used heavily in compilers).
- Memory support, including local stack. Keep in mind pointer aliasing problem. Again, decompilers of JVM and .NET files are simpler here.

16.9 Further reading

There are several interesting open-source attempts to build decompiler. Both source code and theses are interesting study.

- *decomp* by Jim Reuter[11].

[7] This regular expression string hasn't been properly escaped, for the reason of easier readability and understanding.

[8] Floating-point unit

[9] Java Virtual Machine

[10] https://yurichev.com/mirrors/vanEmmerik_ssa.pdf

[11] http://www.program-transformation.org/Transform/DecompReadMe, http://www.program-transformation.org/Transform/DecompDecompiler

- *DCC* by Cristina Cifuentes[12].

  It is interesting that this decompiler supports only one type (*int*). Maybe this is a reason why DCC decompiler produces source code with *.B* extension? Read more about B typeless language (C predecessor): https://yurichev.com/blog/typeless/.

- *Boomerang* by Mike Van Emmerik, Trent Waddington et al[13].

As I've mentioned, LISP knowledge can help to understand this all much easier. Here is well-known micro-interpreter of LISP by Peter Norvig, also written in Python: https://web.archive.org/web/20161116133448/http://www.norvig.com/lispy.html, https://web.archive.org/web/20160305172301/http://norvig.com/lispy2.html.

The gradual rewriting I've shown here is also available in Mathematica (”Trace” command): https://reference.wolfram.com/language/ref/Trace.html.

I also enjoyed reading “The Elements of Artificial Intelligence” book by Steve Tanimoto [14], chapter 3: “Production Systems and Pattern Matching”.

See also: Nuno Lopes – Verifying Optimizations using SMT Solvers [15].

16.10 The files

Python version and tests: https://smt.st/current_tree/toy_decompiler/files.

There are also C and Racket versions, but outdated.

Keep in mind—this decompiler is still at toy level, and it was tested only on tiny test files supplied.

[12] http://www.program-transformation.org/Transform/DccDecompiler, thesis: https://yurichev.com/mirrors/DCC_decompilation_thesis.pdf

[13] http://boomerang.sourceforge.net/, http://www.program-transformation.org/Transform/MikeVanEmmerik, thesis: https://yurichev.com/mirrors/vanEmmerik_ssa.pdf

[14] https://archive.org/details/TheElementsOfArtificialIntelligenceUsingLisp/

[15] https://llvm.org/devmtg/2013-11/slides/Lopes-SMT.pdf

Chapter 17

Symbolic execution

Untitled collection of math notes[1]has short intro to symbolic computation.

17.1 Symbolic execution

17.1.1 Swapping two values using XOR

There is a well-known (but counterintuitive) algorithm for swapping two values in two variables using XOR operation without use of any additional memory/register:

```
X=X^Y
Y=Y^X
X=X^Y
```

How it works? It would be better to construct an expression at each step of execution.

```
#!/usr/bin/env python3

class Expr:
    def __init__(self,s):
        self.s=s

    def __str__(self):
        return self.s

    def __xor__(self, other):
        return Expr("(" + self.s + "^" + other.s + ")")

def XOR_swap(X, Y):
    X=X^Y
    Y=Y^X
    X=X^Y
    return X, Y

new_X, new_Y=XOR_swap(Expr("X"), Expr("Y"))
print ("new_X", new_X)
print ("new_Y", new_Y)
```

It works, because Python is dynamically typed PL, so the function doesn't care what to operate on, numerical values, or on objects of Expr() class.

Here is result:

```
new_X ((X^Y)^(Y^(X^Y)))
new_Y (Y^(X^Y))
```

You can remove double variables in your mind (since XORing by a value twice will result in nothing). At new_X we can drop two X-es and two Y-es, and single Y will left. At new_Y we can drop two Y-es, and single X will left.

[1] https://xn--mxacd.xn--qxam/math-notes.pdf

17.1.2 Change endianness

What does this code do?

```
mov     eax, ecx
mov     edx, ecx
shl     edx, 16
and     eax, 0000ff00H
or      eax, edx
mov     edx, ecx
and     edx, 00ff0000H
shr     ecx, 16
or      edx, ecx
shl     eax, 8
shr     edx, 8
or      eax, edx
```

In fact, many reverse engineers play shell game a lot, keeping track of what is stored where, at each point of time.

Figure 17.1: Hieronymus Bosch – The Conjurer

Again, we can build equivalent function which can take both numerical variables and Expr() objects. We also extend Expr() class to support many arithmetical and boolean operations. Also, Expr() methods would take both Expr() objects on input and integer values.

```
#!/usr/bin/env python3

class Expr:
    def __init__(self,s):
        self.s=s

    def convert_to_Expr_if_int(self, n):
        if isinstance(n, int):
            return Expr(str(n))
        if isinstance(n, Expr):
            return n
        raise AssertionError # unsupported type

    def __str__(self):
        return self.s

    def __xor__(self, other):
        return Expr("(" + self.s + "^" + self.convert_to_Expr_if_int(other).s + ")")

    def __and__(self, other):
        return Expr("(" + self.s + "&" + self.convert_to_Expr_if_int(other).s + ")")

    def __or__(self, other):
        return Expr("(" + self.s + "|" + self.convert_to_Expr_if_int(other).s + ")")

    def __lshift__(self, other):
        return Expr("(" + self.s + "<<" + self.convert_to_Expr_if_int(other).s + ")")

    def __rshift__(self, other):
        return Expr("(" + self.s + ">>" + self.convert_to_Expr_if_int(other).s + ")")

# change endianness
ecx=Expr("initial_ECX") # 1st argument
eax=ecx             #  mov     eax, ecx
edx=ecx             #  mov     edx, ecx
edx=edx<<16         #  shl     edx, 16
eax=eax&0xff00      #  and     eax, 0000ff00H
eax=eax|edx         #  or      eax, edx
edx=ecx             #  mov     edx, ecx
edx=edx&0x00ff0000  #  and     edx, 00ff0000H
ecx=ecx>>16         #  shr     ecx, 16
edx=edx|ecx         #  or      edx, ecx
eax=eax<<8          #  shl     eax, 8
edx=edx>>8          #  shr     edx, 8
eax=eax|edx         #  or      eax, edx

print (eax)
```

I run it:

```
((((initial_ECX&65280)|(initial_ECX<<16))<<8)|(((initial_ECX&16711680)|(initial_ECX
   >>16))>>8))
```

Now this is something more readable, however, a bit LISPy at first sight. In fact, this is a function which change endianness in 32-bit word.

By the way, my Toy Decompiler can do this job as well, but operates on AST instead of plain strings: 16.

A real life story

In past, before Hex-Rays decompiler was available, the author reversed a lot of x86 code simply by rewriting it in the text editor, in the very same manner as I've shown here.

Maybe it was primitive, but easier that trying to guess what is stored in which register at the each moment.

An example of this method: 19.2.7.

17.1.3 Fast Fourier transform

I've found one of the smallest possible FFT implementations on reddit:

```
#!/usr/bin/env python3

```

```
from cmath import exp,pi

def FFT(X):
    n = len(X)
    w = exp(-2*pi*1j/n)
    if n > 1:
        X = FFT(X[::2]) + FFT(X[1::2])
        for k in range(int(n/2)):
            xk = X[k]
            X[k] = xk + w**k*X[int(k+n/2)]
            X[int(k+n/2)] = xk - w**k*X[int(k+n/2)]
    return X

print (FFT([1,2,3,4,5,6,7,8]))
```

Just interesting, what value has each element on output?

```
#!/usr/bin/env python3

from cmath import exp,pi

class Expr:
    def __init__(self,s):
        self.s=s

    def convert_to_Expr_if_int(self, n):
        if isinstance(n, int):
            return Expr(str(n))
        if isinstance(n, Expr):
            return n
        raise AssertionError # unsupported type

    def __str__(self):
        return self.s

    def __add__(self, other):
        return Expr("(" + self.s + "+" + self.convert_to_Expr_if_int(other).s + ")")

    def __sub__(self, other):
        return Expr("(" + self.s + "-" + self.convert_to_Expr_if_int(other).s + ")")

    def __mul__(self, other):
        return Expr("(" + self.s + "*" + self.convert_to_Expr_if_int(other).s + ")")

    def __pow__(self, other):
        return Expr("(" + self.s + "**" + self.convert_to_Expr_if_int(other).s + ")")

def FFT(X):
    n = len(X)
    # cast complex value to string, and then to Expr
    w = Expr(str(exp(-2*pi*1j/n)))
    if n > 1:
        X = FFT(X[::2]) + FFT(X[1::2])
        for k in range(int(n/2)):
            xk = X[k]
            X[k] = xk + w**k*X[int(k+n/2)]
            X[int(k+n/2)] = xk - w**k*X[int(k+n/2)]
    return X

input=[Expr("input_%d" % i) for i in range(8)]
output=FFT(input)
for i in range(len(output)):
    print (i, ":", output[i])
```

FFT() function left almost intact, the only thing I added: complex value is converted into string and then Expr() object is constructed.

```
0 : (((input_0+(((-1-1.22464679915e-16j)**0)*input_4))+(((6.12323399574e-17-1j)**0)*(
    input_2+(((-1-1.22464679915e-16j)**0)*input_6))))+(((0.707106781187-0.707106781187
    j)**0)*((input_1+(((-1-1.22464679915e-16j)**0)*input_5))+(((6.12323399574e-17-1j)
    **0)*(input_3+(((-1-1.22464679915e-16j)**0)*input_7))))))
1 : (((input_0-(((-1-1.22464679915e-16j)**0)*input_4))+(((6.12323399574e-17-1j)**1)*(
    input_2-(((-1-1.22464679915e-16j)**0)*input_6))))+(((0.707106781187-0.707106781187
    j)**1)*((input_1-(((-1-1.22464679915e-16j)**0)*input_5))+(((6.12323399574e-17-1j)
    **1)*(input_3-(((-1-1.22464679915e-16j)**0)*input_7))))))
2 : (((input_0+(((-1-1.22464679915e-16j)**0)*input_4))-(((6.12323399574e-17-1j)**0)*(
    input_2+(((-1-1.22464679915e-16j)**0)*input_6))))+(((0.707106781187-0.707106781187
    j)**2)*((input_1+(((-1-1.22464679915e-16j)**0)*input_5))-(((6.12323399574e-17-1j)
    **0)*(input_3+(((-1-1.22464679915e-16j)**0)*input_7))))))
3 : (((input_0-(((-1-1.22464679915e-16j)**0)*input_4))-(((6.12323399574e-17-1j)**1)*(
    input_2-(((-1-1.22464679915e-16j)**0)*input_6))))+(((0.707106781187-0.707106781187
    j)**3)*((input_1-(((-1-1.22464679915e-16j)**0)*input_5))-(((6.12323399574e-17-1j)
    **1)*(input_3-(((-1-1.22464679915e-16j)**0)*input_7))))))
4 : (((input_0+(((-1-1.22464679915e-16j)**0)*input_4))+(((6.12323399574e-17-1j)**0)*(
    input_2+(((-1-1.22464679915e-16j)**0)*input_6))))-(((0.707106781187-0.707106781187
    j)**0)*((input_1+(((-1-1.22464679915e-16j)**0)*input_5))+(((6.12323399574e-17-1j)
    **0)*(input_3+(((-1-1.22464679915e-16j)**0)*input_7))))))
5 : (((input_0-(((-1-1.22464679915e-16j)**0)*input_4))+(((6.12323399574e-17-1j)**1)*(
    input_2-(((-1-1.22464679915e-16j)**0)*input_6))))-(((0.707106781187-0.707106781187
    j)**1)*((input_1-(((-1-1.22464679915e-16j)**0)*input_5))+(((6.12323399574e-17-1j)
    **1)*(input_3-(((-1-1.22464679915e-16j)**0)*input_7))))))
6 : (((input_0+(((-1-1.22464679915e-16j)**0)*input_4))-(((6.12323399574e-17-1j)**0)*(
    input_2+(((-1-1.22464679915e-16j)**0)*input_6))))-(((0.707106781187-0.707106781187
    j)**2)*((input_1+(((-1-1.22464679915e-16j)**0)*input_5))-(((6.12323399574e-17-1j)
    **0)*(input_3+(((-1-1.22464679915e-16j)**0)*input_7))))))
7 : (((input_0-(((-1-1.22464679915e-16j)**0)*input_4))-(((6.12323399574e-17-1j)**1)*(
    input_2-(((-1-1.22464679915e-16j)**0)*input_6))))-(((0.707106781187-0.707106781187
    j)**3)*((input_1-(((-1-1.22464679915e-16j)**0)*input_5))-(((6.12323399574e-17-1j)
    **1)*(input_3-(((-1-1.22464679915e-16j)**0)*input_7))))))
```

We can see subexpressions in form like x^0 and x^1. We can eliminate them, since $x^0 = 1$ and $x^1 = x$. Also, we can reduce subexpressions like $x \cdot 1$ to just x.

```
    def __mul__(self, other):
        op1=self.s
        op2=self.convert_to_Expr_if_int(other).s

        if op1=="1":
            return Expr(op2)
        if op2=="1":
            return Expr(op1)

        return Expr("(" + op1 + "*" + op2 + ")")

    def __pow__(self, other):
        op2=self.convert_to_Expr_if_int(other).s
        if op2=="0":
            return Expr("1")
        if op2=="1":
            return Expr(self.s)

        return Expr("(" + self.s + "**" + op2 + ")")
```

```
0 : (((input_0+input_4)+(input_2+input_6))+((input_1+input_5)+(input_3+input_7)))
```

```
1 : (((input_0-input_4)+((6.12323399574e-17-1j)*(input_2-input_6)))
    +((0.707106781187-0.707106781187j)*((input_1-input_5)+((6.12323399574e-17-1j)*(
    input_3-input_7)))))
2 : (((input_0+input_4)-(input_2+input_6))+(((0.707106781187-0.707106781187j)**2)*((
    input_1+input_5)-(input_3+input_7))))
3 : (((input_0-input_4)-((6.12323399574e-17-1j)*(input_2-input_6)))
    +(((0.707106781187-0.707106781187j)**3)*((input_1-input_5)-((6.12323399574e-17-1j)
    *(input_3-input_7)))))
4 : (((input_0+input_4)+(input_2+input_6))-((input_1+input_5)+(input_3+input_7)))
5 : (((input_0-input_4)+((6.12323399574e-17-1j)*(input_2-input_6)))
    -((0.707106781187-0.707106781187j)*((input_1-input_5)+((6.12323399574e-17-1j)*(
    input_3-input_7)))))
6 : (((input_0+input_4)-(input_2+input_6))-(((0.707106781187-0.707106781187j)**2)*((
    input_1+input_5)-(input_3+input_7))))
7 : (((input_0-input_4)-((6.12323399574e-17-1j)*(input_2-input_6)))
    -(((0.707106781187-0.707106781187j)**3)*((input_1-input_5)-((6.12323399574e-17-1j)
    *(input_3-input_7)))))
```

17.1.4 Cyclic redundancy check

I've always been wondering, which input bit affects which bit in the final CRC32 value.

From the CRC theory (good and concise introduction: http://web.archive.org/web/20161220015646/http://www.hackersdelight.org/crc.pdf) we know that CRC is shifting register with taps.

We will track each bit rather than byte or word, which is highly inefficient, but serves our purpose better:

```
#!/usr/bin/env python3

import sys

class Expr:
    def __init__(self,s):
        self.s=s

    def convert_to_Expr_if_int(self, n):
        if isinstance(n, int):
            return Expr(str(n))
        if isinstance(n, Expr):
            return n
        raise AssertionError # unsupported type

    def __str__(self):
        return self.s

    def __xor__(self, other):
        return Expr("(" + self.s + "^" + self.convert_to_Expr_if_int(other).s + ")")

BYTES=1

def crc32(buf):
    #state=[Expr("init_%d" % i) for i in range(32)]
    state=[Expr("1") for i in range(32)]
    for byte in buf:
        for n in range(8):
            bit=byte[n]
            to_taps=bit^state[31]
            state[31]=state[30]
            state[30]=state[29]
            state[29]=state[28]
            state[28]=state[27]
            state[27]=state[26]
            state[26]=state[25]^to_taps
            state[25]=state[24]
            state[24]=state[23]
            state[23]=state[22]^to_taps
            state[22]=state[21]^to_taps
            state[21]=state[20]
            state[20]=state[19]
            state[19]=state[18]
            state[18]=state[17]
            state[17]=state[16]
            state[16]=state[15]^to_taps
            state[15]=state[14]
            state[14]=state[13]
            state[13]=state[12]
            state[12]=state[11]^to_taps
            state[11]=state[10]^to_taps
            state[10]=state[9]^to_taps
            state[9]=state[8]
            state[8]=state[7]^to_taps
            state[7]=state[6]^to_taps
            state[6]=state[5]
            state[5]=state[4]^to_taps
            state[4]=state[3]^to_taps
            state[3]=state[2]
            state[2]=state[1]^to_taps
            state[1]=state[0]^to_taps
            state[0]=to_taps

    for i in range(32):
        print ("state %d=%s" % (i, state[31-i]))

buf=[[Expr("in_%d_%d" % (byte, bit)) for bit in range(8)] for byte in range(BYTES)]
crc32(buf)
```

Here are expressions for each CRC32 bit for 1-byte buffer:

```
state 0=(1^(in_0_2^1))
state 1=((1^(in_0_0^1))^(in_0_3^1))
state 2=(((1^(in_0_0^1))^(in_0_1^1))^(in_0_4^1))
state 3=(((1^(in_0_1^1))^(in_0_2^1))^(in_0_5^1))
state 4=(((1^(in_0_2^1))^(in_0_3^1))^(in_0_6^(1^(in_0_0^1))))
state 5=(((1^(in_0_3^1))^(in_0_4^1))^(in_0_7^(1^(in_0_1^1))))
state 6=((1^(in_0_4^1))^(in_0_5^1))
state 7=((1^(in_0_5^1))^(in_0_6^(1^(in_0_0^1))))
state 8=(((1^(in_0_0^1))^(in_0_6^(1^(in_0_0^1))))^(in_0_7^(1^(in_0_1^1))))
state 9=((1^(in_0_1^1))^(in_0_7^(1^(in_0_1^1))))
state 10=(1^(in_0_2^1))
state 11=(1^(in_0_3^1))
state 12=((1^(in_0_0^1))^(in_0_4^1))
state 13=(((1^(in_0_0^1))^(in_0_1^1))^(in_0_5^1))
state 14=((((1^(in_0_0^1))^(in_0_1^1))^(in_0_2^1))^(in_0_6^(1^(in_0_0^1))))
state 15=((((1^(in_0_1^1))^(in_0_2^1))^(in_0_3^1))^(in_0_7^(1^(in_0_1^1))))
state 16=((((1^(in_0_0^1))^(in_0_2^1))^(in_0_3^1))^(in_0_4^1))
state 17=(((((1^(in_0_0^1))^(in_0_1^1))^(in_0_3^1))^(in_0_4^1))^(in_0_5^1))
state 18=(((((1^(in_0_1^1))^(in_0_2^1))^(in_0_4^1))^(in_0_5^1))^(in_0_6^(1^(in_0_0^1)
   )))
state 19=((((((1^(in_0_0^1))^(in_0_2^1))^(in_0_3^1))^(in_0_5^1))^(in_0_6^(1^(in_0_0
   ^1))))^(in_0_7^(1^(in_0_1^1))))
state 20=((((((1^(in_0_0^1))^(in_0_1^1))^(in_0_3^1))^(in_0_4^1))^(in_0_6^(1^(in_0_0
   ^1))))^(in_0_7^(1^(in_0_1^1))))
state 21=(((((1^(in_0_1^1))^(in_0_2^1))^(in_0_4^1))^(in_0_5^1))^(in_0_7^(1^(in_0_1^1)
   )))
state 22=(((((1^(in_0_0^1))^(in_0_2^1))^(in_0_3^1))^(in_0_5^1))^(in_0_6^(1^(in_0_0^1)
    )))
state 23=((((((1^(in_0_0^1))^(in_0_1^1))^(in_0_3^1))^(in_0_4^1))^(in_0_6^(1^(in_0_0
    ^1))))^(in_0_7^(1^(in_0_1^1))))
state 24=((((((in_0_0^1)^(in_0_1^1))^(in_0_2^1))^(in_0_4^1))^(in_0_5^1))^(in_0_7^(1^(
    in_0_1^1))))
state 25=(((((in_0_1^1)^(in_0_2^1))^(in_0_3^1))^(in_0_5^1))^(in_0_6^(1^(in_0_0^1))))
state 26=(((((in_0_2^1)^(in_0_3^1))^(in_0_4^1))^(in_0_6^(1^(in_0_0^1))))^(in_0_7^(1^(
    in_0_1^1))))
state 27=((((in_0_3^1)^(in_0_4^1))^(in_0_5^1))^(in_0_7^(1^(in_0_1^1))))
state 28=(((in_0_4^1)^(in_0_5^1))^(in_0_6^(1^(in_0_0^1))))
state 29=(((in_0_5^1)^(in_0_6^(1^(in_0_0^1))))^(in_0_7^(1^(in_0_1^1))))
state 30=((in_0_6^(1^(in_0_0^1)))^(in_0_7^(1^(in_0_1^1))))
state 31=(in_0_7^(1^(in_0_1^1)))
```

For larger buffer, expressions gets increasing exponentially. This is 0th bit of the final state for 4-byte buffer:

```
state 0=(((((((((((((in_0_0^1)^(in_0_1^1))^(in_0_2^1))^(in_0_4^1))^(in_0_5^1))^(
    in_0_7^(1^(in_0_1^1))))^
(in_1_0^(1^(in_0_2^1))))^(in_1_2^(((1^(in_0_0^1))^(in_0_1^1))^(in_0_4^1))))^(in_1_3
    ^(((1^(in_0_1^1))^
(in_0_2^1))^(in_0_5^1))))^(in_1_4^(((1^(in_0_2^1))^(in_0_3^1))^(in_0_6^(1^(in_0_0^1))
    ))))^(in_2_0^((((1^
(in_0_0^1))^(in_0_6^(1^(in_0_0^1))))^(in_0_7^(1^(in_0_1^1))))^(in_1_2^(((1^(in_0_0^1)
    )^(in_0_1^1))^(in_0_4^
1))))))^(in_2_6^((((((1^(in_0_0^1))^(in_0_1^1))^(in_0_2^1))^(in_0_6^(1^(in_0_0^1))))
    ^(in_1_4^(((1^(in_0_2^1))^
(in_0_3^1))^(in_0_6^(1^(in_0_0^1))))))^(in_1_5^(((1^(in_0_3^1))^(in_0_4^1))^(in_0_7
    ^(1^(in_0_1^1))))))^
(in_2_0^((((1^(in_0_0^1))^(in_0_6^(1^(in_0_0^1))))^(in_0_7^(1^(in_0_1^1))))^(in_1_2
    ^(((1^(in_0_0^1))^(in_0_1^1))^
(in_0_4^1))))))))^(in_2_7^(((((((1^(in_0_1^1))^(in_0_2^1))^(in_0_3^1))^(in_0_7^(1^(
    in_0_1^1))))^(in_1_5^(((1^
(in_0_3^1))^(in_0_4^1))^(in_0_7^(1^(in_0_1^1))))))^(in_1_6^(((1^(in_0_4^1))^(in_0_5
    ^1))^(in_1_0^(1^(in_0_2^
1))))))^(in_2_1^((((1^(in_0_1^1))^(in_0_7^(1^(in_0_1^1))))^(in_1_0^(1^(in_0_2^1))))^(
    in_1_3^(((1^(in_0_1^1))^
(in_0_2^1))^(in_0_5^1))))))))^(in_3_2^((((((((1^(in_0_1^1))^(in_0_2^1))^(in_0_4^1))
    ^(in_0_5^1))^(in_0_6^(1^
(in_0_0^1))))^(in_1_2^(((1^(in_0_0^1))^(in_0_1^1))^(in_0_4^1))))^(in_2_0^((((1^(
    in_0_0^1))^(in_0_6^(1^(in_0_0^
1))))^(in_0_7^(1^(in_0_1^1))))^(in_1_2^(((1^(in_0_0^1))^(in_0_1^1))^(in_0_4^1))))))^(
    in_2_1^((((1^(in_0_1^1))^
(in_0_7^(1^(in_0_1^1))))^(in_1_0^(1^(in_0_2^1))))^(in_1_3^(((1^(in_0_1^1))^(in_0_2^1)
    )^(in_0_5^1))))))^(in_2_4^
(((((1^(in_0_0^1))^(in_0_4^1))^(in_1_2^(((1^(in_0_0^1))^(in_0_1^1))^(in_0_4^1))))^(
    in_1_3^(((1^(in_0_1^1))^
(in_0_2^1))^(in_0_5^1))))^(in_1_6^(((1^(in_0_4^1))^(in_0_5^1))^(in_1_0^(1^(in_0_2^1))
    ))))))))
```

Expression for the 0th bit of the final state for 8-byte buffer has length of $\approx 350KiB$, which is, of course, can be reduced significantly (because this expression is basically XOR tree), but you can feel the weight of it.

Now we can process this expressions somehow to get a smaller picture on what is affecting what. Let's say, if we can find "in_2_3" substring in expression, this means that 3rd bit of 2nd byte of input affects this expression. But even more than that: since this is XOR tree (i.e., expression consisting only of XOR operations), if some input variable is occurring twice, it's *annihilated*, since $x \oplus x = 0$. More than that: if a variable occurred even number of times (2, 4, 8, etc), it's annihilated, but left if it's occurred odd number of times (1, 3, 5, etc).

```
    for i in range(32):
        #print "state %d=%s" % (i, state[31-i])
        sys.stdout.write ("state %02d: " % i)
        for byte in range(BYTES):
            for bit in range(8):
                s="in_%d_%d" % (byte, bit)
                if str(state[31-i]).count(s) & 1:
                    sys.stdout.write ("*")
                else:
                    sys.stdout.write (" ")
        sys.stdout.write ("\n")
```

(https://smt.st/current_tree/symbolic/exec/4_CRC/2.py)

Now this how each bit of 1-byte input buffer affects each bit of the final CRC32 state:

```
state 00:   *
state 01: *  *
state 02: **  *
state 03:  **  *
state 04: * **  *
state 05:  * **  *
state 06:     **
state 07: *    **
state 08:  *    **
state 09:        *
state 10:   *
state 11:    *
state 12: *   *
state 13: **   *
state 14:  **   *
state 15:   **   *
state 16: * ***
state 17: ** ***
state 18: *** ***
state 19:  *** ***
state 20:    ** **
state 21:   * ** *
state 22:   ** **
state 23:    ** **
state 24: * * ** *
state 25: **** **
state 26: ***** **
state 27:  * *** *
state 28: *   ***
state 29: **   ***
state 30: **    **
state 31:  *     *
```

This is 8*8=64 bits of 8-byte input buffer:

```
state 00:  * ** *  ***  * ** **      *  * ***** ***       *   * **  *
state 01: * * ** *  ***  * ** **      *  * ***** ***       *   * **  *
state 02: ** * ** *  ***  * ** **      *  * ***** ***       *   * **  *
state 03: *** * ** *  ***  * ** **      *  * ***** ***       *   * **  *
state 04: **** * ** *  ***  * ** **      *  * ***** ***       *   * **  *
state 05:  **** * ** *  ***  * ** **      *  * ***** ***       *   * **  *
state 06:  **  ***   ** **   *  ** ***  * * **     ** *** *   *  *    **
state 07: * **  ***   ** **   *  ** ***  * * **     ** *** *   *  *    **
state 08:  * **  ***   ** **   *  ** ***  * * **     ** *** *   *  *    **
state 09:  *** ** *  *   ** *** *  *****  * * ** **   ** * * ** *        *
state 10:  **    *  *** *      * *  * **  * * ** * *   **   *  **   *
state 11:   **    *  *** *      * *  * **  * * ** * *   **   *  **   *
state 12:    **    *  *** *      * *  * **  * * ** * *   **   *  **   *
state 13:     **    *  *** *      * *  * **  * * ** * *   **   *  **   *
state 14:      **    *  *** *      * *  * **  * * ** * *   **   *  **   *
state 15:       **    *  *** *      * *  * **  * * ** * *   **   *  **   *
state 16:  * ** ****** **   **         **  *  *  *  ** * **  *  *** ***
state 17: * * ** ****** **   **         **  *  *  *  ** * **  *  *** ***
state 18:  * * ** ****** **   **         **  *  *  *  ** * **  *  *** ***
state 19: * * * ** ****** **   **         **  *  *  *  ** * **  *  *** ***
state 20:     ******  ** ** *** **   *  * *  *****   *  **** *  *    ** **
state 21: ** *** **  * *       * **  ** *** ** *      *  * **   *   * ** *
state 22:   ** *  * ***   ** ** * ** *****  *    **    *    *** *   ** **
state 23: *  ** *  * ***   ** ** * ** *****  *    **    *    *** *   ** **
state 24:    * *** * ***  *** *** * *  * *  **  *****    **    * ** * ** *
state 25:  * *   *** ***  * * **** *       **   *  ***     *  *  ***** **
state 26: * * *   *** ***  * * **** *       **   *  ***     *  *  ***** **
state 27: *   ***      * *****  ****    * ***   **   ***  *  **  * * *** *
state 28:    *** * ***      *    *****  ***   * *     *** **   ****   ***
state 29:     *** * ***      *    *****  ***   * *     *** **   ****   ***
state 30: ** *** *  * *** ** *     ** ***    **  *      **  *** * **    **
state 31: * ** *  ***  * ** **      *  * ***** ***       *   * **  *     *
```

17.1.5 Linear congruential generator

This is popular PRNG from OpenWatcom CRT[2] library: https://github.com/open-watcom/open-watcom-v2/blob/d468b609ba6ca61eeddad80dd2485e3256fc5261/bld/clib/math/c/rand.c.

What expression it generates on each step?

```
#!/usr/bin/env python3

class Expr:
    def __init__(self,s):
        self.s=s

    def convert_to_Expr_if_int(self, n):
        if isinstance(n, int):
            return Expr(str(n))
        if isinstance(n, Expr):
            return n
        raise AssertionError # unsupported type

    def __str__(self):
        return self.s

    def __xor__(self, other):
        return Expr("(" + self.s + "^" + self.convert_to_Expr_if_int(other).s + ")")

    def __mul__(self, other):
        return Expr("(" + self.s + "*" + self.convert_to_Expr_if_int(other).s + ")")

    def __add__(self, other):
        return Expr("(" + self.s + "+" + self.convert_to_Expr_if_int(other).s + ")")

    def __and__(self, other):
        return Expr("(" + self.s + "&" + self.convert_to_Expr_if_int(other).s + ")")

    def __rshift__(self, other):
        return Expr("(" + self.s + ">>" + self.convert_to_Expr_if_int(other).s + ")")

seed=Expr("initial_seed")

def rand():
    global seed
    seed=seed*1103515245+12345
    return (seed>>16) & 0x7fff

for i in range(10):
```

[2] C runtime library

```
    print (i, ":", rand())
```

```
0 : ((((initial_seed*1103515245)+12345)>>16)&32767)
1 : ((((((initial_seed*1103515245)+12345)*1103515245)+12345)>>16)&32767)
2 : ((((((((initial_seed*1103515245)+12345)*1103515245)+12345)*1103515245)+12345)
    >>16)&32767)
3 : ((((((((((initial_seed*1103515245)+12345)*1103515245)+12345)*1103515245)+12345)
    *1103515245)+12345)>>16)&32767)
4 : ((((((((((((initial_seed*1103515245)+12345)*1103515245)+12345)*1103515245)+12345)
    *1103515245)+12345)*1103515245)+12345)>>16)&32767)
5 : ((((((((((((((initial_seed*1103515245)+12345)*1103515245)+12345)*1103515245)
    +12345)*1103515245)+12345)*1103515245)+12345)*1103515245)+12345)>>16)&32767)
6 : ((((((((((((((((initial_seed*1103515245)+12345)*1103515245)+12345)*1103515245)
    +12345)*1103515245)+12345)*1103515245)+12345)*1103515245)+12345)*1103515245)
    +12345)>>16)&32767)
7 : ((((((((((((((((((initial_seed*1103515245)+12345)*1103515245)+12345)*1103515245)
    +12345)*1103515245)+12345)*1103515245)+12345)*1103515245)+12345)*1103515245)
    +12345)*1103515245)+12345)>>16)&32767)
8 : ((((((((((((((((((((initial_seed*1103515245)+12345)*1103515245)+12345)
    *1103515245)+12345)*1103515245)+12345)*1103515245)+12345)*1103515245)+12345)
    *1103515245)+12345)*1103515245)+12345)*1103515245)+12345)>>16)&32767)
9 : ((((((((((((((((((((((initial_seed*1103515245)+12345)*1103515245)+12345)
    *1103515245)+12345)*1103515245)+12345)*1103515245)+12345)*1103515245)+12345)
    *1103515245)+12345)*1103515245)+12345)*1103515245)+12345)*1103515245)+12345)>>16)
    &32767)
```

Now if we once got several values from this PRNG, like 4583, 16304, 14440, 32315, 28670, 12568..., how would we recover the initial seed? The problem in fact is solving a system of equations:

```
((((initial_seed*1103515245)+12345)>>16)&32767)==4583
((((((initial_seed*1103515245)+12345)*1103515245)+12345)>>16)&32767)==16304
((((((((initial_seed*1103515245)+12345)*1103515245)+12345)*1103515245)+12345)>>16)
    &32767)==14440
((((((((((initial_seed*1103515245)+12345)*1103515245)+12345)*1103515245)+12345)
    *1103515245)+12345)>>16)&32767)==32315
```

As it turns out, Z3 can solve this system correctly using only two equations:

```
#!/usr/bin/env python3

from z3 import *

s=Solver()

x=BitVec("x",32)

a=1103515245
c=12345
s.add((((x*a)+c)>>16)&32767==4583)
s.add((((((x*a)+c)*a)+c)>>16)&32767==16304)
#s.add((((((((x*a)+c)*a)+c)*a)+c)>>16)&32767==14440)
#s.add((((((((((x*a)+c)*a)+c)*a)+c)*a)+c)>>16)&32767==32315)

s.check()
print (s.model())
```

```
[x = 11223344]
```

(Though, it takes ≈ 20 seconds on my ancient Intel Atom netbook.)

17.1.6 Path constraint

Example 1

How to get weekday from UNIX timestamp?

```
#!/usr/bin/env python

input=...
SECS_DAY=24*60*60
dayno = input / SECS_DAY
wday = (dayno + 4) % 7
if wday==5:
    print "Thanks God, it's Friday!"
```

Let's say, we should find a way to run the block with print() call in it. What input value should be?

First, let's build expression of *wday* variable:

```
#!/usr/bin/env python3

class Expr:
    def __init__(self,s):
        self.s=s

    def convert_to_Expr_if_int(self, n):
        if isinstance(n, int):
            return Expr(str(n))
        if isinstance(n, Expr):
            return n
        raise AssertionError # unsupported type

    def __str__(self):
        return self.s

    def __truediv__(self, other):
        return Expr("(" + self.s + "/" + self.convert_to_Expr_if_int(other).s + ")")

    def __mod__(self, other):
        return Expr("(" + self.s + "%" + self.convert_to_Expr_if_int(other).s + ")")

    def __add__(self, other):
        return Expr("(" + self.s + "+" + self.convert_to_Expr_if_int(other).s + ")")

    def __eq__(self, other):
        return Expr("(" + self.s + "==" + self.convert_to_Expr_if_int(other).s + ")")

input=Expr("input")
SECS_DAY=24*60*60
dayno = input / SECS_DAY
wday = (dayno + 4) % 7
cond=(wday==5)
print ("cond:", cond)
```

```
cond: ((((input/86400)+4)%7)==5)
```

In order to execute the block, we should solve this equation: $((\frac{input}{86400} + 4) \equiv 5 \mod 7$.

So far, this is easy task for Z3:

```
#!/usr/bin/env python3

from z3 import *

s=Solver()
```

```
x=Int("x")
s.add(x>0)

dayno=Int("dayno")
s.add(dayno==x/86400)
s.add((dayno+4)%7==5) # must be Friday

s.check()
print (s.model())
```

```
[x = 86400, dayno = 1]
```

This is indeed correct UNIX timestamp for Friday:

```
% date -d @86400 -u "+%a %Y-%m-%d %H:%M:%S"
Fri 1970-01-02 00:00:00
```

Though the date back in year 1970, but it's still correct!

This is also called "path constraint", i.e., what constraint must be satisfied to execute specific block? Several tools has "path" in their names, like "pathgrind", Symbolic PathFinder, CodeSurfer Path Inspector, etc.

Like the shell game, this task is also often encounters in practice. You can see that something dangerous can be executed inside some basic block and you're trying to deduce, what input values can cause execution of it. It may be buffer overflow, etc. Such input values are sometimes also called "inputs of death".

Many crackmes are solved in this way, all you need is find a path into block which prints "key is correct" or something like that.

We can extend this tiny example:

```
input=...
SECS_DAY=24*60*60
dayno = input / SECS_DAY
wday = (dayno + 4) % 7
print wday
if wday==5:
    print "Thanks God, it's Friday!"
else:
    print "Got to wait a little"
```

Now we have two blocks: for the first we should solve this equation: $((\frac{input}{86400} + 4) \equiv 5 \mod 7$. But for the second we should solve inverted equation: $((\frac{input}{86400} + 4) \not\equiv 5 \mod 7$. By solving these equations, we will find two paths into both blocks.

KLEE (or similar tool) tries to find path to each [basic] block and produces "ideal" unit test. Hence, KLEE can find a path into the block which crashes everything, or reporting about correctness of the input key/license, etc. Surprisingly, KLEE can find backdoors in the very same manner.

KLEE is also called "KLEE Symbolic Virtual Machine" – by that its creators mean that the KLEE is VM[3] which executes a code symbolically rather than numerically (like usual CPU).

How KLEE would handle this example?

```
#include <stdint.h>
#include <assert.h>

int is_it_friday(uint32_t input)
{
        uint32_t SECS_DAY=24*60*60;
        uint32_t dayno = input / SECS_DAY;
        uint32_t wday = (dayno + 4) % 7;
        if (wday==5)
                return 1; // friday
        else
                return 0;
};
```

[3] Virtual Machine

```
int main()
{
    uint32_t input;
    klee_make_symbolic(&input, sizeof input, "input");
    if (is_it_friday(input))
            klee_assert (0);
};
```

KLEE found two inputs, the one for ’return 1’ and the second for ’return 0’:

```
% clang -emit-llvm -c -g -O0 -Xclang -disable-O0-optnone -I /tmp/klee_src/include/
   klee/ TGIF_KLEE1.c

% time klee --libc=uclibc --optimize --use-query-log=solver:smt2 TGIF_KLEE1.bc

...
KLEE: ERROR: TGIF_KLEE1.c:20: failed external call: klee_assert
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 11734
KLEE: done: completed paths = 1
KLEE: done: partially completed paths = 1
KLEE: done: generated tests = 2
...

% ls klee-last/*
klee-last/assembly.ll   klee-last/run.stats                 klee-last/test000001.ktest
klee-last/info          klee-last/solver-queries.smt2       klee-last/test000002.ktest
klee-last/messages.txt  klee-last/test000001.external.err   klee-last/warnings.txt
klee-last/run.istats    klee-last/test000001.kquery

% ktest-tool klee-last/test000001.ktest | grep uint
object 0: uint: 86400

% ktest-tool klee-last/test000002.ktest | grep uint
object 0: uint: 0
```

So two inputs: 0 and 86400. One for not-Friday, another is for Friday:

```
% date -d @86400 -u "+%a %Y-%m-%d %H:%M:%S"
Fri 1970-01-02 00:00:00

% date -d @0 -u "+%a %Y-%m-%d %H:%M:%S"
Thu 1970-01-01 00:00:00
```

KLEE can produce SMT-LIB file for SMT-solver, if the ’–use-query-log=solver:smt2’ option supplied: ’solver-queries.smt2’. This is excerpt:

```
; Query 1 -- Type: InitialValues, Instructions: 11658
(set-option :produce-models true)
(set-logic QF_AUFBV )
(declare-fun input () (Array (_ BitVec 32) (_ BitVec 8) ) )
(assert (=  (_ bv5 32) (bvurem  (bvadd  (_ bv4 32) (bvudiv  (concat  (select  input (
   _ bv3 32) ) (concat  (select  input (_ bv2 32) ) (concat  (select  input (_ bv1
   32) ) (select  input (_ bv0 32) ) ) ) ) (_ bv86400 32) )
) (_ bv7 32) ) ) )
(check-sat)
(get-value ( (select input (_ bv0 32) ) ) )
(get-value ( (select input (_ bv1 32) ) ) )
(get-value ( (select input (_ bv2 32) ) ) )
(get-value ( (select input (_ bv3 32) ) ) )
```

It divided 32-bit uint to 4 bytes. Let’s run this in Z3:

```
z3 -smt2 solver-queries.smt2_excerpt
sat
(((select input (_ bv0 32) ) #x80))
(((select input (_ bv1 32) ) #xc6))
(((select input (_ bv2 32) ) #x41))
(((select input (_ bv3 32) ) #xc8))
```

0xc841c680=3359753856.

It's still friday, but in future:

```
% date -d @3359753856 -u "+%a %Y-%m-%d %H:%M:%S"
Fri 2076-06-19 00:57:36
```

And CBMC?

```
#include <stdint.h>
#include <assert.h>

int is_it_friday(uint32_t input)
{
        uint32_t SECS_DAY=24*60*60;
        uint32_t dayno = input / SECS_DAY;
        uint32_t wday = (dayno + 4) % 7;
        if (wday==5)
                return 1; // friday
        else
                return 0;
};

void check(uint64_t input)
{
        assert (is_it_friday(input)==0);
};
```

I'll ask CBMC to find such an input so that assert() would trigger:

```
% cbmc --trace --function check TGIF_CBMC1.c

...

State 21 file TGIF_CBMC1.c function __CPROVER__start line 15 thread 0
----------------------------------------------------
  INPUT input: 86401ul (00000000 00000000 00000000 00000000 00000000 00000001
      01010001 10000001)

...

Violated property:
  file TGIF_CBMC1.c function check line 17 thread 0
  assertion is_it_friday(input)==0
  return_value_is_it_friday == 0
```

This value is for Friday again:

```
% date -d @86401 -u "+%a %Y-%m-%d %H:%M:%S"
Fri 1970-01-02 00:00:01
```

CBMC can also produce an SMT-LIB 2.x file for a SMT solver, with this problem converted to SMT-LIB 2.x:

```
% cbmc --smt2 --outfile TGIF_CBMC1_dest.smt --trace --function check TGIF_CBMC1.c
```

It's not very verbose, thanks to the tiny size of our problem:

Listing 17.1: Excerpt

```
; set_to true (equal)
(define-fun |is_it_friday::1::SECS_DAY!0@1#2| () (_ BitVec 32) (_ bv86400 32))

; set_to true (equal)
(define-fun |is_it_friday::1::dayno!0@1#2| () (_ BitVec 32) (bvudiv |is_it_friday::
    input!0@1#1| (_ bv86400 32)))

; set_to true (equal)
(define-fun |is_it_friday::1::wday!0@1#2| () (_ BitVec 32) (bvurem (bvadd (_ bv4 32)
    |is_it_friday::1::dayno!0@1#2|) (_ bv7 32)))
```

Full version: https://smt.st/current_tree/symbolic/exec/6_TGIF/TGIF_CBMC1_dest.smt.

I can’t stand the urge and I’ll execute this using Z3:

```
% z3 -smt2 TGIF_CBMC1_dest.smt
```

(https://smt.st/current_tree/symbolic/exec/6_TGIF/TGIF_CBMC1_dest.smt.result)

This is the computed *input* value:

```
...
((|__CPROVER__start::input!0@1#2| #x00000000b1796100))
...
```

0xb1796100=2977521920

Still, this is Friday, but the Year is 2064:

```
% date -d @2977521920 -u "+%a %Y-%m-%d %H:%M:%S"
Fri 2064-05-09 01:25:20
```

By the way, if the ’–z3’ option used with CBMC, it will spawn Z3 with that file.

Example 2

Let’s make our example more realistic. We can go to beer pub within some time range, say, between 18:00 and 00:00.

```
#!/usr/bin/env python3

class Expr:
    def __init__(self,s):
        self.s=s

    def convert_to_Expr_if_int(self, n):
        if isinstance(n, int):
            return Expr(str(n))
        if isinstance(n, Expr):
            return n
        raise AssertionError # unsupported type

    def __str__(self):
        return self.s

    def __truediv__(self, other):
        return Expr("(" + self.s + "/" + self.convert_to_Expr_if_int(other).s + ")")

    def __mod__(self, other):
        return Expr("(" + self.s + "%" + self.convert_to_Expr_if_int(other).s + ")")

    def __add__(self, other):
        return Expr("(" + self.s + "+" + self.convert_to_Expr_if_int(other).s + ")")

    def __eq__(self, other):
        return Expr("(" + self.s + "==" + self.convert_to_Expr_if_int(other).s + ")")

    def __ge__(self, other):
        return Expr("(" + self.s + ">=" + self.convert_to_Expr_if_int(other).s + ")")

    def __le__(self, other):
        return Expr("(" + self.s + "<=" + self.convert_to_Expr_if_int(other).s + ")")

input=Expr("input")
SECS_DAY=24*60*60
dayno = input / SECS_DAY
wday = (dayno + 4) % 7
cond1=(wday==5)
print ("cond1:", cond1)

# also check hour
seconds_within_day=input % SECS_DAY
hour=seconds_within_day / 3600
cond2=hour>=18
cond3=hour<=23
print ("cond2:", cond2)
print ("cond3:", cond3)
```

Now we have 3 conditions in system of equations:

```
cond1: ((((input/86400)+4)%7)==5)
cond2: (((input%86400)/3600)>=18)
cond3: (((input%86400)/3600)<=23)
```

It's an easy problem for Z3:

```
#!/usr/bin/env python3

from z3 import *

s=Solver()

x=Int("x")
s.add(x>0)

# 1st constraint:
dayno=Int("dayno")
s.add(dayno==x/86400)
s.add((dayno+4)%7==5) # must be Friday

# 2nd constraint:
hour=Int("hour")
s.add(hour==(x%86400)/3600) # get hour (UTC time)
s.add(hour>=18)
s.add(hour<=23)

s.check()
print (s.model())
```

```
[x = 154799, hour = 18, dayno = 1]
```

UTC time, but correct:

```
% date -d @154799 -u "+%a %Y-%m-%d %H:%M:%S"
Fri 1970-01-02 18:59:59
```

Reworked KLEE example:

```
#include <stdint.h>
#include <assert.h>

int lets_party(uint32_t input)
```

```
{
        uint32_t SECS_HOUR=60*60;
        uint32_t SECS_DAY=24*SECS_HOUR;
        uint32_t dayno = input / SECS_DAY;
        uint32_t wday = (dayno + 4) % 7;
        if (wday==5)
        {
                // friday

                uint32_t hour=(input % SECS_DAY) / SECS_HOUR;

                if (hour>=18 && hour<=23)
                        return 1;
                else
                        return 0;
        }
        else
                return 0;
};

int main()
{
    uint32_t input;
    klee_make_symbolic(&input, sizeof input, "input");
    if (lets_party(input))
            klee_assert (0);
};
```

KLEE finds 3 paths:

```
% time klee --libc=uclibc --optimize --use-query-log=solver:smt2 TGIF_KLEE2.bc

...
KLEE: done: total instructions = 11820
KLEE: done: completed paths = 2
KLEE: done: partially completed paths = 1
KLEE: done: generated tests = 3
...

% ls klee-last/*
klee-last/assembly.ll   klee-last/run.stats                  klee-last/test000001.ktest
klee-last/info          klee-last/solver-queries.smt2        klee-last/test000002.ktest
klee-last/messages.txt  klee-last/test000001.external.err  klee-last/test000003.ktest
klee-last/run.istats    klee-last/test000001.kquery          klee-last/warnings.txt

% ktest-tool klee-last/test000001.ktest | grep uint
object 0: uint: 159744

% ktest-tool klee-last/test000002.ktest | grep uint
object 0: uint: 0

% ktest-tool klee-last/test000003.ktest | grep uint
object 0: uint: 86400
```

So there are 3 values KLEE suggests:

- 0 (not friday)
- 86400 (just friday)
- 159744 (friday plus hour in needed range)

```
% date -d @159744 -u "+%a %Y-%m-%d %H:%M:%S"
Fri 1970-01-02 20:22:24
```

What SMT file KLEE prepares? Now it consists of two SMT queries. First for first 'if' statement, second is for second:

```
; Query 1 -- Type: InitialValues, Instructions: 11656
(set-option :produce-models true)
(set-logic QF_AUFBV )
(declare-fun input () (Array (_ BitVec 32) (_ BitVec 8) ) )
(assert (=  (_ bv5 32) (bvurem  (bvadd  (_ bv4 32) (bvudiv  (concat  (select  input (
   _ bv3 32) ) (concat  (select  input (_ bv2 32) ) (concat  (select  input (_ bv1
   32) ) (select  input (_ bv0 32) ) ) ) ) (_ bv86400 32) ) ) (_ bv7 32) ) ) )
(check-sat)
(get-value ( (select input (_ bv0 32) ) ) )
(get-value ( (select input (_ bv1 32) ) ) )
(get-value ( (select input (_ bv2 32) ) ) )
(get-value ( (select input (_ bv3 32) ) ) )
(exit)
;   OK -- Elapsed: 2.303087e-02s
;   Solvable: true
;     input = [128,81,1,0]

; Query 2 -- Type: InitialValues, Instructions: 11676
(set-option :produce-models true)
(set-logic QF_AUFBV )
(declare-fun input () (Array (_ BitVec 32) (_ BitVec 8) ) )
(assert (let ( (?B1 (concat  (select  input (_ bv3 32) ) (concat  (select  input (_
   bv2 32) ) (concat  (select  input (_ bv1 32) ) (select  input (_ bv0 32) ) ) ) ) )
    ) (and  (bvult  (_ bv64799 32) (bvurem  ?B1 (_ bv86400 32) ) ) (=  (_ bv5 32) (
   bvurem  (bvadd  (_ bv4 32) (bvudiv  ?B1 (_ bv86400 32) ) ) (_ bv7 32) ) ) ) ) )
(check-sat)
(get-value ( (select input (_ bv0 32) ) ) )
(get-value ( (select input (_ bv1 32) ) ) )
(get-value ( (select input (_ bv2 32) ) ) )
(get-value ( (select input (_ bv3 32) ) ) )
(exit)
;   OK -- Elapsed: 2.374635e-02s
;   Solvable: true
;     input = [0,112,2,0]
```

Now reworked example for CBMC:

```
#include <stdint.h>
#include <assert.h>

int is_it_friday(uint32_t input)
{
        uint32_t SECS_DAY=24*60*60;
        uint32_t dayno = input / SECS_DAY;
        uint32_t wday = (dayno + 4) % 7;
        if (wday==5)
                return 1; // friday
        else
                return 0;
};

void check(uint64_t input)
{
        assert (is_it_friday(input)==0);
};
```

```
% cbmc --trace --function check TGIF_CBMC2.c

...

** Results:
TGIF_CBMC2.c function check
[check.assertion.1] line 27 assertion lets_party(input)==0: FAILURE

Trace for check.assertion.1:

State 21 file TGIF_CBMC2.c function __CPROVER__start line 25 thread 0
----------------------------------------------------
  INPUT input: 165600ul (00000000 00000000 00000000 00000000 00000000 00000010
      10000110 11100000)
...
```

```
% date -d @165600 -u "+%a %Y-%m-%d %H:%M:%S"
Fri 1970-01-02 22:00:00
```

The SMT-LIB file generated by CBMC:

Listing 17.2: Excerpt

```
; set_to true (equal)
(define-fun |lets_party::1::SECS_HOUR!0@1#2| () (_ BitVec 32) (_ bv3600 32))

; set_to true (equal)
(define-fun |lets_party::1::SECS_DAY!0@1#2| () (_ BitVec 32) (_ bv86400 32))

; set_to true (equal)
(define-fun |lets_party::1::dayno!0@1#2| () (_ BitVec 32) (bvudiv |lets_party::input
    !0@1#1| (_ bv86400 32)))

; set_to true (equal)
(define-fun |lets_party::1::wday!0@1#2| () (_ BitVec 32) (bvurem (bvadd (_ bv4 32) |
    lets_party::1::dayno!0@1#2|) (_ bv7 32)))

; set_to true (equal)
(define-fun |lets_party::1::1::hour!0@1#2| () (_ BitVec 32) (bvudiv (bvurem |
    lets_party::input!0@1#1| (_ bv86400 32)) (_ bv3600 32)))
```

Summary

In short, this is how KLEE and CBMC work. They construct a system of equations for each basic block and then ask SMT solver to find correct inputs.

17.1.7 Division by zero

If division by zero is unwrapped by sanitizing check, and exception isn't caught, it can crash process.

Let's calculate simple expression $\frac{x}{2y+4z-12}$. We can add a warning into `__div__` method:

```
#!/usr/bin/env python3

class Expr:
    def __init__(self,s):
        self.s=s

    def convert_to_Expr_if_int(self, n):
        if isinstance(n, int):
            return Expr(str(n))
        if isinstance(n, Expr):
            return n
        raise AssertionError # unsupported type

    def __str__(self):
        return self.s

    def __mul__(self, other):
        return Expr("(" + self.s + "*" + self.convert_to_Expr_if_int(other).s + ")")

    def __truediv__(self, other):
        op2=self.convert_to_Expr_if_int(other).s
        print ("warning: division by zero if "+op2+"==0")
        return Expr("(" + self.s + "/" + op2 + ")")

    def __add__(self, other):
        return Expr("(" + self.s + "+" + self.convert_to_Expr_if_int(other).s + ")")

    def __sub__(self, other):
        return Expr("(" + self.s + "-" + self.convert_to_Expr_if_int(other).s + ")")

"""
     x
------------
2y + 4z - 12

"""

def f(x, y, z):
    return x/(y*2 + z*4 - 12)

print (f(Expr("x"), Expr("y"), Expr("z")))
```

...so it will report about dangerous states and conditions:

```
warning: division by zero if (((y*2)+(z*4))-12)==0
(x/(((y*2)+(z*4))-12))
```

This equation is easy to solve, let's try Wolfram Mathematica this time:

```
In[]:= FindInstance[{(y*2 + z*4) - 12 == 0}, {y, z}, Integers]
Out[]= {{y -> 0, z -> 3}}
```

These values for y and z can also be called "inputs of death".

17.1.8 Merge sort

How merge sort works? I have copypasted Python code from rosettacode.com almost intact:

```
#!/usr/bin/env python3

class Expr:
    def __init__(self,s,i):
        self.s=s
        self.i=i

    def __str__(self):
        # return both symbolic and integer:
        return self.s+" (" + str(self.i)+")"

    def __le__(self, other):
        # compare only integer parts:
        return self.i <= other.i
```

```
# copypasted from http://rosettacode.org/wiki/Sorting_algorithms/Merge_sort#Python
def merge(left, right):
    result = []
    left_idx, right_idx = 0, 0
    while left_idx < len(left) and right_idx < len(right):
        # change the direction of this comparison to change the direction of the sort
        if left[left_idx] <= right[right_idx]:
            result.append(left[left_idx])
            left_idx += 1
        else:
            result.append(right[right_idx])
            right_idx += 1

    if left_idx < len(left):
        result.extend(left[left_idx:])
    if right_idx < len(right):
        result.extend(right[right_idx:])
    return result

def tabs (t):
    return "\t"*t

def merge_sort(m, indent=0):
    print (tabs(indent)+"merge_sort() begin. input:")
    for i in m:
        print (tabs(indent)+str(i))

    if len(m) <= 1:
        print (tabs(indent)+"merge_sort() end. returning single element")
        return m

    middle = len(m) // 2
    left = m[:middle]
    right = m[middle:]

    left = merge_sort(left, indent+1)
    right = merge_sort(right, indent+1)
    rt=list(merge(left, right))
    print (tabs(indent)+"merge_sort() end. returning:")
    for i in rt:
        print (tabs(indent)+str(i))
    return rt

# input buffer has both symbolic and numerical values:
input=[Expr("input1",22), Expr("input2",7), Expr("input3",2), Expr("input4",1), Expr(
    "input5",8), Expr("input6",4)]
merge_sort(input)
```

But here is a function which compares elements. Obviously, it wouldn't work correctly without it.

So we can track both expression for each element and numerical value. Both will be printed finally. But whenever values are to be compared, only numerical parts will be used.

Result:

```
merge_sort() begin. input:
input1 (22)
input2 (7)
input3 (2)
input4 (1)
input5 (8)
input6 (4)
        merge_sort() begin. input:
        input1 (22)
        input2 (7)
        input3 (2)
                merge_sort() begin. input:
                input1 (22)
                merge_sort() end. returning single element
                merge_sort() begin. input:
                input2 (7)
                input3 (2)
                        merge_sort() begin. input:
                        input2 (7)
                        merge_sort() end. returning single element
                        merge_sort() begin. input:
                        input3 (2)
                        merge_sort() end. returning single element
                merge_sort() end. returning:
                input3 (2)
                input2 (7)
        merge_sort() end. returning:
        input3 (2)
        input2 (7)
        input1 (22)
        merge_sort() begin. input:
        input4 (1)
        input5 (8)
        input6 (4)
                merge_sort() begin. input:
                input4 (1)
                merge_sort() end. returning single element
                merge_sort() begin. input:
                input5 (8)
                input6 (4)
                        merge_sort() begin. input:
                        input5 (8)
                        merge_sort() end. returning single element
                        merge_sort() begin. input:
                        input6 (4)
                        merge_sort() end. returning single element
                merge_sort() end. returning:
                input6 (4)
                input5 (8)
        merge_sort() end. returning:
        input4 (1)
        input6 (4)
        input5 (8)
merge_sort() end. returning:
input4 (1)
input3 (2)
input6 (4)
input2 (7)
input5 (8)
input1 (22)
```

17.1.9 Extending Expr class

This is somewhat senseless, nevertheless, it's easy task to extend my Expr class to support AST instead of plain strings. It's also possible to add folding steps (like I demonstrated in Toy Decompiler: 16). Maybe someone will want to do this as an exercise. By the way, the toy decompiler can be used as a simple symbolic engine as well, just feed all the instructions to it and it will track contents of each register.

17.1.10 Conclusion

For the sake of demonstration, I made things plain, as simple as possible. But reality is always harsh and inconvenient, so all this shouldn't be taken as a silver bullet.

The files used in this part: https://smt.st/current_tree/symbolic/exec.

17.2 Further reading

- Robert W. Floyd — Assigning meaning to programs [4].
- James C. King — Symbolic Execution and Program Testing [5].
- History of symbolic execution (as well as SAT/SMT solving, fuzzing, and taint data tracking) [6].

17.3 Tools

- http://angr.io - static and dynamic symbolic ("concolic") analysis.

17.4 Examples

- https://fevral.github.io/2017/08/13/flareon2015-2.html – using angr.io.
- Breaking Kryptonite's obfuscation: a static analysis approach relying on symbolic execution [7].
- Sean Heelan – Anatomy of a Symbolic Emulator[8]

[4] https://classes.soe.ucsc.edu/cmps290g/Fall09/Papers/AssigningMeanings1967.pdf

[5] https://yurichev.com/mirrors/king76symbolicexecution.pdf

[6] https://github.com/enzet/symbolic-execution

[7] https://doar-e.github.io/blog/2013/09/16/breaking-kryptonites-obfuscation-with-symbolic-execution/

[8] https://sean.heelan.io/2012/03/23/anatomy-of-a-symbolic-emulator-part-1-trace-generation/, https://sean.heelan.io/2012/03/23/anatomy-of-a-symbolic-emulator-part-2-introducing-symbolic-data/, https://sean.heelan.io/2012/03/23/anatomy-of-a-symbolic-emulator-part-3-processing-symbolic-data-generating-new-inputs/

Chapter 18

KLEE

18.1 Installation

KLEE building from source is tricky. Easiest way to use KLEE is to install *docker* and then to run KLEE docker image[1]. To copy files to/from docker, use the “docker cp” command.

18.2 Unit test: HTML/CSS color

The most popular ways to represent HTML/CSS color is by English name (e.g., “red”) and by 6-digit hexadecimal number (e.g., “#0077CC”).

There is a third, less popular way: if each byte in hexadecimal number has two doubling digits, it can be *abbreviated*, thus, “#0077CC” can be written just as “#07C”.

Let’s write a function that identify a way of color encoding. 1) color name (if possible, first priority); 2) 3-digit hexadecimal form (if possible, second priority); 3) or as 6-digit hexadecimal form (as a last resort).

```
#include <string.h>
#include <stdio.h>
#include <stdint.h>
#include <assert.h>

#include "klee.h"

int HTML_color(uint8_t R, uint8_t G, uint8_t B)
{
        if (R==0 && G==0 && B==0) // "black"
                return 1;

        if (R==0xFF && G==0 && B==0) // "red"
                return 2;

        if (R==0x0 && G==0xFF && B==0) // "green"
                return 3;

        if (R==0 && G==0 && B==0xFF) // "blue"
                return 4;

        // abbreviated hexadecimal
        // like "#123"
        if ((R>>4)==(R&0xF) && (G>>4)==(G&0xF) && (B>>4)==(B&0xF))
                return 5;

        // last resort
        // like "#123456"
        return 6;
};
```

[1] http://klee.github.io/docker/

```
int main()
{
        uint8_t R, G, B;
        klee_make_symbolic (&R, sizeof R, "R");
        klee_make_symbolic (&G, sizeof R, "G");
        klee_make_symbolic (&B, sizeof R, "B");

        // without passing return value out of main(), KLEE wouldn't work correctly
        // perhaps, this is how we show that the return value is used somehow
        // instead of dangling as void
        return HTML_color(R, G, B);
};
```

There are 6 possible paths in function, and let's see, if KLEE could find them all? It's indeed so:

```
% clang -emit-llvm -c -g -O0 -Xclang -disable-O0-optnone -I /tmp/klee_src/include/
   klee/ color.c

% klee --libc=uclibc --optimize color.bc
...
KLEE: done: total instructions = 12232
KLEE: done: completed paths = 8
KLEE: done: partially completed paths = 0
KLEE: done: generated tests = 8
...
```

So there are 8 paths:

```
% ls klee-last
assembly.ll   run.istats           test000001.ktest  test000004.ktest  test000007.
   ktest
info          run.stats            test000002.ktest  test000005.ktest  test000008.
   ktest
messages.txt  solver-queries.smt2  test000003.ktest  test000006.ktest  warnings.txt
```

Let's list all 8 test cases:

```
% ktest-tool klee-last/test000001.ktest | grep hex
object 0: hex : 0x00
object 1: hex : 0x00
object 2: hex : 0x00
% ktest-tool klee-last/test000002.ktest | grep hex
object 0: hex : 0xff
object 1: hex : 0x00
object 2: hex : 0x00
% ktest-tool klee-last/test000003.ktest | grep hex
object 0: hex : 0x00
object 1: hex : 0xff
object 2: hex : 0x00
% ktest-tool klee-last/test000004.ktest | grep hex
object 0: hex : 0x00
object 1: hex : 0x00
object 2: hex : 0xff
% ktest-tool klee-last/test000005.ktest | grep hex
object 0: hex : 0x01
object 1: hex : 0x01
object 2: hex : 0x01
% ktest-tool klee-last/test000006.ktest | grep hex
object 0: hex : 0x00
object 1: hex : 0x00
object 2: hex : 0x01
% ktest-tool klee-last/test000007.ktest | grep hex
```

```
object 0: hex : 0x00
object 1: hex : 0x01
object 2: hex : 0x00
% ktest-tool klee-last/test000008.ktest | grep hex
object 0: hex : 0x11
object 1: hex : 0x00
object 2: hex : 0x00
```

1st test for black, 2nd/3rd/4th tests — red/green/blue.

5th/6th/7th tests — *full* 6-nibble hex code.

8th test — *abbreviated* 3-nibble hex code.

The test set could be a bit shorter (6 against 8), but this is not bad too.

18.3 Unit test: strcmp() function

The standard `strcmp()` function from C library can return 0, -1 or 1, depending of comparison result.

Here is my own implementation of `strcmp()`:

```
#include "klee.h"

int my_strcmp(const char *s1, const char *s2)
{
        int ret = 0;

        while (1)
        {
                ret = *(unsigned char *) s1 - *(unsigned char *) s2;
                if (ret!=0)
                        break;
                if ((*s1==0) || (*s2)==0)
                        break;
                s1++;
                s2++;
        };

        if (ret < 0)
        {
                return -1;
        } else if (ret > 0)
        {
                return 1;
        }

        return 0;
}

int main()
{
        char input1[2];
        char input2[2];

        klee_make_symbolic(input1, sizeof input1, "input1");
        klee_make_symbolic(input2, sizeof input2, "input2");

        klee_assume((input1[0]>='a') && (input1[0]<='z'));
        klee_assume((input2[0]>='a') && (input2[0]<='z'));

        klee_assume(input1[1]==0);
        klee_assume(input2[1]==0);

        // without passing return value out of main(), KLEE wouldn't work correctly
        // perhaps, this is how we show that the return value is used somehow
        // instead of dangling as void
        return my_strcmp (input1, input2);
};
```

Let's find out, if KLEE is capable of finding all three paths? I intentionally made things simpler for KLEE by limiting input arrays to two 2 bytes or to 1 character + terminal zero byte.

```
% clang -emit-llvm -c -g -O0 -Xclang -disable-O0-optnone -I /tmp/klee_src/include/
   klee/ strcmp.c

% klee --libc=uclibc --optimize strcmp.bc
KLEE: done: completed paths = 3
KLEE: done: partially completed paths = 0
KLEE: done: generated tests = 3

% ls klee-last
assembly.ll  messages.txt  run.stats          test000002.ktest  warnings.txt
info         run.istats    test000001.ktest   test000003.ktest
```

These are the input values for each path inside of my implementation of `strcmp()`:

```
% ktest-tool klee-last/test000001.ktest | grep data
object 0: data: b'a\x00'
object 1: data: b'c\x00'

% ktest-tool klee-last/test000002.ktest | grep data
object 0: data: b'a\x00'
object 1: data: b'a\x00'

% ktest-tool klee-last/test000003.ktest | grep data
object 0: data: b'c\x00'
object 1: data: b'a\x00'
```

1st is if first argument ("a") is lesser than the second ("c"), 2nd is if they are equal ("a" and "a"), 3rd is fi first argument ("c") is greater than the second ("a").

Using these 3 test cases, we've got full coverage of our (toy) implementation of `strcmp()`.

18.4 UNIX date/time

UNIX date/time is a number of seconds that have elapsed since 1-Jan-1970 00:00 UTC. C/C++ gmtime() function is used to decode this value into human-readable date/time.

Here is a piece of code I've copypasted from some ancient version of Minix OS[2] and reworked slightly:

```
#include <stdio.h>
#include <stdint.h>
#include <time.h>
#include <assert.h>

#include "klee.h"

/*
 * copypasted and reworked from
 * https://web.archive.org/web/20170413231318/http://www.cise.ufl.edu/~cop4600/cgi-
    bin/lxr/http/source.cgi/lib/ansi/loc_time.h
 * https://web.archive.org/web/20170114033506/http://www.cise.ufl.edu/~cop4600/cgi-
    bin/lxr/http/source.cgi/lib/ansi/misc.c
 * https://web.archive.org/web/20170413223528/http://www.cise.ufl.edu/~cop4600/cgi-
    bin/lxr/http/source.cgi/lib/ansi/gmtime.c
 */
```

[2] https://web.archive.org/web/20170413223528/http://www.cise.ufl.edu/~cop4600/cgi-bin/lxr/http/source.cgi/lib/ansi/gmtime.c

```

#define YEAR0           1900
#define EPOCH_YR        1970
#define SECS_DAY        (24L * 60L * 60L)
#define YEARSIZE(year)  (LEAPYEAR(year) ? 366 : 365)

const int _ytab[2][12] =
{
        { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 },
        { 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
};

const char *_days[] =
{
        "Sunday", "Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "Saturday"
};

const char *_months[] =
{
        "January", "February", "March",
        "April", "May", "June",
        "July", "August", "September",
        "October", "November", "December"
};

#define LEAPYEAR(year)  (!((year) % 4) && (((year) % 100) || !((year) % 400)))

void decode_UNIX_time(const time_t time)
{
        unsigned int dayclock, dayno;
        int year = EPOCH_YR;

        dayclock = (unsigned long)time % SECS_DAY;
        dayno = (unsigned long)time / SECS_DAY;

        int seconds = dayclock % 60;
        int minutes = (dayclock % 3600) / 60;
        int hour = dayclock / 3600;
        int wday = (dayno + 4) % 7;
        while (dayno >= YEARSIZE(year))
        {
                dayno -= YEARSIZE(year);
                year++;
        }

        year = year - YEAR0;

        int month = 0;

        while (dayno >= _ytab[LEAPYEAR(year)][month])
        {
                dayno -= _ytab[LEAPYEAR(year)][month];
                month++;
        }

        char *s;
        switch (month)
        {
                case 0: s="January"; break;
                case 1: s="February"; break;
```

```
 75                 case 2: s="March"; break;
 76                 case 3: s="April"; break;
 77                 case 4: s="May"; break;
 78                 case 5: s="June"; break;
 79                 case 6: s="July"; break;
 80                 case 7: s="August"; break;
 81                 case 8: s="September"; break;
 82                 case 9: s="October"; break;
 83                 case 10: s="November"; break;
 84                 case 11: s="December"; break;
 85                 default:
 86                          assert(0);
 87         };
 88
 89         printf ("%04d-%s-%02d %02d:%02d:%02d\n", YEAR0+year, s, dayno+1, hour,
               minutes, seconds);
 90         printf ("week day: %s\n", _days[wday]);
 91 }
 92
 93 int main()
 94 {
 95         uint32_t time;
 96
 97         klee_make_symbolic(&time, sizeof time, "time");
 98
 99         decode_UNIX_time(time);
100
101         return 0;
102 }
```

Let's try it[3].

```
% clang -emit-llvm -c -g -O0 -Xclang -disable-O0-optnone -I /tmp/klee_src/include/
   klee/ klee_time1.c

...

% time klee --libc=uclibc --optimize klee_time1.bc
KLEE: ERROR: klee_time1.c:89: external call with symbolic argument: printf
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: klee_time1.c:86: ASSERTION FAIL: 0
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 68809
KLEE: done: completed paths = 0
KLEE: done: partially completed paths = 1635
KLEE: done: generated tests = 2

real    1m42.781s
user    1m32.635s
...
```

Wow, assert() at line 86 has been triggered, why? Let's see a value of UNIX time which triggers it:

```
% ls klee-last
assembly.ll   run.istats           test000001.exec.err  test000002.assert.err
   warnings.txt
info          run.stats            test000001.kquery    test000002.kquery
messages.txt  solver-queries.smt2  test000001.ktest     test000002.ktest

% ktest-tool klee-last/test000002.ktest | grep uint
```

[3] All timings are when KLEE 3.0 running on AMD Ryzen 5 3600.

```
object 0: uint: 978243712
```

Let's decode this value using UNIX date utility:

```
% date -u --date='@978243712'
Sun Dec 31 06:21:52 AM UTC 2000
```

After my investigation, I've found that `month` variable can hold incorrect value of 12 (while 11 is maximal, for December), because LEAPYEAR() macro must receive year number as 2000, not as 100.

Correct fix would be moving year substraction expression down:

```
% diff -u klee_time1.c klee_time1_fixed.c
--- klee_time1.c        2023-12-04 18:25:34.077200826 +0200
+++ klee_time1_fixed.c  2023-12-04 18:48:28.352408590 +0200
@@ -57,8 +57,6 @@
                year++;
        }

-       year = year - YEAR0;
-
        int month = 0;

        while (dayno >= _ytab[LEAPYEAR(year)][month])
@@ -66,6 +64,8 @@
                dayno -= _ytab[LEAPYEAR(year)][month];
                month++;
        }
+
+       year = year - YEAR0;
```

So I've introduced a bug during rewriting this function, and KLEE found it! Or was it bug in the original code?

Just interesting, what would be if I'll replace switch() to array of strings, like it usually happens in concise C/C++ code?

```
% diff -u klee_time1.c klee_time2.c
--- klee_time1.c        2023-12-04 18:25:34.077200826 +0200
+++ klee_time2.c        2023-12-04 18:34:34.388646376 +0200
@@ -67,24 +67,7 @@
                month++;
        }

-       char *s;
-       switch (month)
-       {
-               case 0: s="January"; break;
-               case 1: s="February"; break;
-               case 2: s="March"; break;
-               case 3: s="April"; break;
-               case 4: s="May"; break;
-               case 5: s="June"; break;
-               case 6: s="July"; break;
-               case 7: s="August"; break;
-               case 8: s="September"; break;
-               case 9: s="October"; break;
-               case 10: s="November"; break;
-               case 11: s="December"; break;
-               default:
-                        assert(0);
-       };
+        const char *s=_months[month];
```

KLEE detects attempt to read beyond array boundaries:

```
% time klee --libc=uclibc --optimize klee_time2.bc
```

```
...
KLEE: ERROR: klee_time2.c:72: external call with symbolic argument: printf
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: klee_time2.c:70: memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 69094
KLEE: done: completed paths = 0
KLEE: done: partially completed paths = 1635
KLEE: done: generated tests = 2

real    1m44.795s
user    1m33.958s
...
```

This is the same UNIX time value we've already seen:

```
% ls klee-last | grep err
test000001.exec.err
test000002.ptr.err

% ktest-tool klee-last/test000002.ktest | grep uint
object 0: uint: 978243712
```

So, if this piece of code can be triggered on remote computer, with this input value (*input of death*), it's possible to crash the process (with some luck, though).

OK, now I'm fixing a bug by moving year subtracting expression down, and let's find, what UNIX time value corresponds to some fancy date like 2022-February-2?

```
...

        while (dayno >= _ytab[LEAPYEAR(year)][month])
        {
                dayno -= _ytab[LEAPYEAR(year)][month];
                month++;
        }
        year = year - YEAR0;

        if (YEAR0+year==2022 && month==1 && dayno+1==22)
                assert(0);
}
```

```
% klee --libc=uclibc --optimize klee_time3.bc
...

KLEE: ERROR: klee_time3.c:49: ASSERTION FAIL: 0
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 175170
KLEE: done: completed paths = 1634
KLEE: done: partially completed paths = 1
KLEE: done: generated tests = 1635

real    0m56.272s
user    0m51.168s

% ls klee-last | grep err
test000569.assert.err

% ktest-tool klee-last/test000569.ktest | grep uint
```

```
object 0: uint: 1645488640

% date -u --date='@1645488640'
Tue Feb 22 00:10:40 UTC 2022
```

Success, but hours/minutes/seconds are seems random—they are random indeed, because KLEE satisfied all constraints we've put, nothing else.

We didn't ask it to set hours/minutes/seconds to zeroes.

Let's add constraints to hours/minutes/seconds as well:

```
        if (YEAR0+year==2022 && month==1 && dayno+1==22 && hour==22 && minutes==22 &&
            seconds==22)
                klee_assert(0);
```

Let's run it and check ...

```
% ls klee-last | grep err
test000578.assert.err

% ktest-tool klee-last/test000578.ktest | grep uint
object 0: uint: 1645568542

% date -u --date='@1645568542'
Tue Feb 22 22:22:22 UTC 2022
```

Now that is precise.

Yes, of course, C/C++ libraries has function(s) to encode human-readable date into UNIX time value, but what we've got here is KLEE working *antipode* of decoding function, *inverse function* in a way.

18.5 Inverse function for base64 decoder

It's piece of cake for KLEE to reconstruct input base64 string given just base64 decoder code without corresponding encoder code. I've copypasted this piece of code from http://www.opensource.apple.com/source/QuickTimeStreamingServer/QuickTimeStreamingServer-452/CommonUtilitiesLib/base64.c.

We add constraints (lines 86, 87) so that output buffer must have byte values from 0 to 15. We also tell to KLEE that the Base64decode() function must return 16 (i.e., size of output buffer in bytes, line 84).

```
 1  #include <string.h>
 2  #include <stdint.h>
 3  #include <stdbool.h>
 4
 5  #include "klee.h"
 6
 7  // copypasted from http://www.opensource.apple.com/source/QuickTimeStreamingServer/
        QuickTimeStreamingServer-452/CommonUtilitiesLib/base64.c
 8
 9  static const unsigned char pr2six[256] =
10  {
11          /* ASCII table */
12          64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
13          64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
14          64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 62, 64, 64, 64, 63,
15          52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 64, 64, 64, 64, 64, 64,
16          64,  0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14,
17          15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 64, 64, 64, 64, 64,
18          64, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
19          41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 64, 64, 64, 64, 64,
20          64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
21          64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
22          64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
23          64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
24          64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
25          64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
        64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
        64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64
};

int Base64decode(char *bufplain, const char *bufcoded)
{
        int nbytesdecoded;
        register const unsigned char *bufin;
        register unsigned char *bufout;
        register int nprbytes;

        bufin = (const unsigned char *) bufcoded;
        while (pr2six[*(bufin++)] <= 63);
        nprbytes = (bufin - (const unsigned char *) bufcoded) - 1;
        nbytesdecoded = ((nprbytes + 3) / 4) * 3;

        bufout = (unsigned char *) bufplain;
        bufin = (const unsigned char *) bufcoded;

        while (nprbytes > 4) {
                *(bufout++) =
                        (unsigned char) (pr2six[*bufin] << 2 | pr2six[bufin[1]] >> 4);
                *(bufout++) =
                        (unsigned char) (pr2six[bufin[1]] << 4 | pr2six[bufin[2]] >> 2);
                *(bufout++) =
                        (unsigned char) (pr2six[bufin[2]] << 6 | pr2six[bufin[3]]);
                bufin += 4;
                nprbytes -= 4;
        }

        /* Note: (nprbytes == 1) would be an error, so just ignore that case */
        if (nprbytes > 1) {
                *(bufout++) =
                        (unsigned char) (pr2six[*bufin] << 2 | pr2six[bufin[1]] >> 4);
        }
        if (nprbytes > 2) {
                *(bufout++) =
                        (unsigned char) (pr2six[bufin[1]] << 4 | pr2six[bufin[2]] >> 2);
        }
        if (nprbytes > 3) {
                *(bufout++) =
                        (unsigned char) (pr2six[bufin[2]] << 6 | pr2six[bufin[3]]);
        }

        *(bufout++) = '\0';
        nbytesdecoded -= (4 - nprbytes) & 3;
        return nbytesdecoded;
}

int main()
{
        char input[32];
        uint8_t output[16+1];

        klee_make_symbolic(input, sizeof input, "input");

        klee_assume(input[31]==0);
```

```
83
84          klee_assume (Base64decode(output, input)==16);
85
86          for (int i=0; i<16; i++)
87                  klee_assume (output[i]==i);
88
89          klee_assert(0);
90
91          return 0;
92  }
```

```
% clang -emit-llvm -c -g -O0 -Xclang -disable-O0-optnone -I /tmp/klee_src/include/
    klee/ klee_base64.c
...

% klee --libc=uclibc --optimize --use-query-log=solver:smt2 klee_base64.bc
KLEE: ERROR: klee_base64.c:84: invalid klee_assume call (provably false)
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: klee_base64.c:89: ASSERTION FAIL: 0
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: klee_base64.c:70: memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: klee_base64.c:66: memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: klee_base64.c:50: memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 19251
KLEE: done: completed paths = 0
KLEE: done: partially completed paths = 32
KLEE: done: generated tests = 5
...
```

We're interesting in the second error, where `klee_assert()` has been triggered:

```
% ls klee-last | grep err
test000001.user.err
test000002.assert.err
test000003.ptr.err
test000004.ptr.err
test000005.ptr.err

% ktest-tool klee-last/test000002.ktest
ktest file : 'klee-last/test000002.ktest'
args       : ['klee_base64.bc']
num objects: 1
object 0: name: 'input'
object 0: size: 32
object 0: data: b'AAECAwQFBgcICQoLDA0OD+\x00\xff\xff\xff\xff\xff\xff\xff\xff\x00'
object 0: hex : 0x41414543417751464267634943516f4c4441304f442b00ffffffffffffffff00
object 0: text: AAECAwQFBgcICQoLDA0OD+..........
```

This is indeed a real base64 string, terminated with the zero byte, just as it's requested by C/C++ standards. The final zero byte at 31th byte (starting at zeroth byte) is our deed: so that KLEE would report lesser number of errors.

The base64 string is indeed correct:

```
% echo AAECAwQFBgcICQoLDA0OD+ | base64 -d | xxd -g 1
base64: invalid input
00000000: 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f  ................
```

base64 decoder Linux utility I've just run blaming for "invalid input"—it means the input string is not properly padded. Now let's pad it manually, and decoder utility will no complain anymore:

```
% echo AAECAwQFBgcICQoLDA0OD+== | base64 -d | xxd -g 1
00000000: 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f  ................
```

The reason our generated base64 string is not padded is because base64 decoders are usually discards padding symbols ("=") at the end. In other words, they are not require them, so is the case of our decoder. Hence, padding symbols are left unnoticed to KLEE.

So we again made *antipode* or *inverse function* of base64 decoder.

18.6 LZSS decompressor

I've googled for a very simple LZSS decompressor and landed at this page:
https://web.archive.org/web/20151202051931/http://www.opensource.apple.com/source/boot/boot-132/i386/boot2/lzss.c

Let's pretend, we're looking at unknown compressing algorithm with no compressor available. Will it be possible to reconstruct a compressed piece of data so that decompressor would generate data we need?

This is my first experiment:

```
// copypasted from https://web.archive.org/web/20151202051931/http://www.opensource.
   apple.com/source/boot/boot-132/i386/boot2/lzss.c

#include <string.h>
#include <stdint.h>
#include <stdbool.h>

#define N         4096  /* size of ring buffer - must be power of 2 */
#define F         18    /* upper limit for match_length */
#define THRESHOLD 2     /* encode string into position and length
                           if match_length is greater than this */
#define NIL       N     /* index for root of binary search trees */

int
decompress_lzss(uint8_t *dst, uint8_t *src, uint32_t srclen)
{
    /* ring buffer of size N, with extra F-1 bytes to aid string comparison */
    uint8_t *dststart = dst;
    uint8_t *srcend = src + srclen;
    int  i, j, k, r, c;
    unsigned int flags;
    uint8_t text_buf[N + F - 1];

    dst = dststart;
    srcend = src + srclen;
    for (i = 0; i < N - F; i++)
        text_buf[i] = ' ';
    r = N - F;
    flags = 0;
    for ( ; ; ) {
        if (((flags >>= 1) & 0x100) == 0) {
            if (src < srcend) c = *src++; else break;
            flags = c | 0xFF00;  /* uses higher byte cleverly */
        }   /* to count eight */
        if (flags & 1) {
            if (src < srcend) c = *src++; else break;
            *dst++ = c;
            text_buf[r++] = c;
            r &= (N - 1);
        } else {
            if (src < srcend) i = *src++; else break;
            if (src < srcend) j = *src++; else break;
            i |= ((j & 0xF0) << 4);
            j = (j & 0x0F) + THRESHOLD;
            for (k = 0; k <= j; k++) {
                c = text_buf[(i + k) & (N - 1)];
                *dst++ = c;
                text_buf[r++] = c;
                r &= (N - 1);
            }
        }
    }

    return dst - dststart;
}

// https://en.wikipedia.org/wiki/
   Buffalo_buffalo_Buffalo_buffalo_buffalo_buffalo_Buffalo_buffalo
#define PLAIN_TEXT "Buffalo buffalo Buffalo"
#define PLAIN_TEXT_LEN strlen(PLAIN_TEXT)

int main()
{
#define COMPRESSED_LEN 15
//#define COMPRESSED_LEN 14 -- fail
        uint8_t input[COMPRESSED_LEN];
        uint8_t plain[PLAIN_TEXT_LEN];
        uint32_t size=COMPRESSED_LEN;

        klee_make_symbolic(input, sizeof input, "input");

        decompress_lzss(plain, input, size);

        for (int i=0; i<PLAIN_TEXT_LEN; i++)
                klee_assume (plain[i]==PLAIN_TEXT[i]);

        klee_assert(0);

        return 0;
}
```

I decreased COMPRESSED_LEN gradually to check, whether KLEE would find compressed piece of data, and it did:

```
% clang -emit-llvm -c -g -O0 -Xclang -disable-O0-optnone -I /tmp/klee_src/include/
   klee/ klee_lzss1.c
...

% time klee --libc=uclibc --optimize klee_lzss1.bc
KLEE: ERROR: klee_lzss1.c:73: invalid klee_assume call (provably false)
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: klee_lzss1.c:46: memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: klee_lzss1.c:36: memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location
KLEE: WARNING ONCE: skipping fork (memory cap exceeded)
KLEE: WARNING: killing 2660 states (over memory cap: 2110MB)
KLEE: WARNING ONCE: calling external: klee_assert(0) at klee_lzss1.c:77 2
KLEE: ERROR: klee_lzss1.c:75: failed external call: klee_assert
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 16853770
KLEE: done: completed paths = 0
KLEE: done: partially completed paths = 184070
KLEE: done: generated tests = 2664
```

```
real    6m31.177s
user    5m43.183s
sys     0m47.280s

% ls klee-last/*err*
klee-last/test000001.user.err  klee-last/test000003.ptr.err
klee-last/test000002.ptr.err   klee-last/test002664.external.err

% ktest-tool klee-last/test002664.ktest
ktest file : 'klee-last/test002664.ktest'
args       : ['klee_lzss1.bc']
num objects: 1
object 0: name: 'input'
object 0: size: 15
object 0: data: b'\xffBuffalo \tb\xef\xf4\xee\xf4'
object 0: hex : 0xff42756666616c6f200962eff4eef4
object 0: text: .Buffalo .b....
```

KLEE consumed $\approx 2GB$ of RAM and worked for $\approx$ 6 minutes (on AMD Ryzen 5 3600).

And here is it, a 15 bytes which, if decompressed by our copypasted algorithm, will result in desired plain text!

So how LZSS works? Without peeking into Wikipedia, we can say that: if LZSS compressor observes some data it already had, it replaces the data with a link to some place in past with size. If it observes something yet unseen, it puts data as is. This is theory. This is indeed what we've got. Desired text is three "Buffalo" words, the first and the last are equivalent, but the second is *almost* equivalent, differing with first by one character.

That's what we see (better formatted C-string):

```
"\xff" "Buffalo " "\x09" "b" "\xef\xf4\xee\xf5"
```

Here is some control byte (0xff), "Buffalo" word is placed *as is*, then another control byte (0x09), then we see beginning of the second word ("b") and more control bytes, perhaps, links to the beginning of the buffer. These are command to decompressor, like, in plain English, "copy data from the buffer we've already done, from that place to that place", etc.

What we've got here is KLEE acting like a LZSS compressor – very slow, complex and expensive, but compressor.

Interesting, is it possible to meddle into this piece of compressed data? Out of whim, can we force KLEE to find a compressed data, where not just "b" character has been placed *as is*, but also the second character of the word, i.e., "bu"?

I've modified main() function by adding `klee_assume()`: now the 11th byte of input (compressed) data (right after "b" byte) must have "u". I has no luck with 15 byte of compressed data, so I increased it to 16 bytes:

```
int main()
{
#define COMPRESSED_LEN 16
        uint8_t input[COMPRESSED_LEN];
        uint8_t plain[PLAIN_TEXT_LEN];
        uint32_t size=COMPRESSED_LEN;

        klee_make_symbolic(input, sizeof input, "input");

        klee_assume(input[11]=='u');

        decompress_lzss(plain, input, size);

        for (int i=0; i<PLAIN_TEXT_LEN; i++)
                klee_assume (plain[i]==PLAIN_TEXT[i]);

        klee_assert(0);

        return 0;
}
```

...et voilà: KLEE found a compressed piece of data which satisfied our whimsical constraint:

```
% time klee --libc=uclibc --optimize klee_lzss2.bc
```

```
...

% ls klee-last/*err*
klee-last/test000001.user.err  klee-last/test000003.ptr.err
klee-last/test000002.ptr.err   klee-last/test002468.external.err

% ktest-tool klee-last/test002468.ktest
ktest file : 'klee-last/test002468.ktest'
args       : ['klee_lzss2.bc']
num objects: 1
object 0: name: 'input'
object 0: size: 16
object 0: data: b'\xffBuffalo \x13bu\xf0\xf2\xed\xf5'
object 0: hex : 0xff42756666616c6f20136275f0f2edf5
object 0: text: .Buffalo .bu....
```

So now we found a piece of compressed data where two strings are placed *as is*: "Buffalo" and "bu".

Both pieces of compressed data, if feeded into our copypasted function, produce "Buffalo buffalo Buffalo" text string.

Please note, I still have no access to LZSS compressor code, and I didn't get into LZSS decompressor details yet. (However, yes, it must be very simple.)

Unfortunately, things are not that cool: larger texts are out of reach for KLEE.

Nevertheless, it's very impressive, taking into account the fact that we're not getting into internals of this specific LZSS decompressor. Once more time, we've created *antipode* of decompressor, or *inverse function*.

Also, as it seems, KLEE isn't very good so far with decompression algorithms (but who's good then?).

18.7 strtodx() from RetroBSD

Just found this function in RetroBSD: https://github.com/RetroBSD/retrobsd/blob/master/src/libc/stdlib/strtod.c. It converts a string into floating point number for given radix.

```
#include <stdio.h>

#include "klee.h"

// my own version, only for radix 10:
int isdigitx (char c, int radix)
{
        if (c>='0' && c<='9')
                return 1;
        return 0;
};

/*
 * double strtodx (char *string, char **endPtr, int radix)
 *      This procedure converts a floating-point number from an ASCII
 *      decimal representation to internal double-precision format.
 *
 * Original sources taken from 386bsd and modified for variable radix
 * by Serge Vakulenko, <vak@kiae.su>.
 *
 * Arguments:
 * string
 *      A decimal ASCII floating-point number, optionally preceded
 *      by white space. Must have form "-I.FE-X", where I is the integer
 *      part of the mantissa, F is the fractional part of the mantissa,
 *      and X is the exponent.  Either of the signs may be "+", "-", or
 *      omitted.  Either I or F may be omitted, or both.  The decimal point
 *      isn't necessary unless F is present. The "E" may actually be an "e",
 *      or "E", "S", "s", "F", "f", "D", "d", "L", "l".
 *      E and X may both be omitted (but not just one).
 *
 * endPtr
 *      If non-NULL, store terminating character's address here.
 *
 * radix
 *      Radix of floating point, one of 2, 8, 10, 16.
 *
 * The return value is the double-precision floating-point
 * representation of the characters in string.  If endPtr isn't
 * NULL, then *endPtr is filled in with the address of the
 * next character after the last one that was part of the
 * floating-point number.
 */
double strtodx (char *string, char **endPtr, int radix)
{
        int sign = 0, expSign = 0, fracSz, fracOff, i;
        double fraction, dblExp, *powTab;
        register char *p;
        register char c;

        /* Exponent read from "EX" field. */
        int exp = 0;

        /* Exponent that derives from the fractional part.  Under normal
         * circumstances, it is the negative of the number of digits in F.
         * However, if I is very long, the last digits of I get dropped
         * (otherwise a long I with a large negative exponent could cause an
         * unnecessary overflow on I alone).  In this case, fracExp is
         * incremented one for each dropped digit. */
        int fracExp = 0;

        /* Number of digits in mantissa. */
        int mantSize;

        /* Number of mantissa digits BEFORE decimal point. */
        int decPt;

        /* Temporarily holds location of exponent in string. */
        char *pExp;

        /* Largest possible base 10 exponent.
         * Any exponent larger than this will already
         * produce underflow or overflow, so there's
         * no need to worry about additional digits. */
        static int maxExponent = 307;

        /* Table giving binary powers of 10.
         * Entry is 10^2^i.  Used to convert decimal
         * exponents into floating-point numbers. */
        static double powersOf10[] = {
                1e1, 1e2, 1e4, 1e8, 1e16, 1e32, //1e64, 1e128, 1e256,
        };
        static double powersOf2[] = {
                2, 4, 16, 256, 65536, 4.294967296e9, 1.8446744073709551616e19,
                //3.4028236692093846346e38, 1.1579208923731619542e77, 1.3407807929942597099e154,
        };
        static double powersOf8[] = {
                8, 64, 4096, 2.81474976710656e14, 7.9228162514264337593e28,
                //6.2771017353866807638e57, 3.9402006196394479212e115,
                    1.5525180923007089351e231,
        };
        static double powersOf16[] = {
                16, 256, 65536, 1.8446744073709551616e19,
                //3.4028236692093846346e38, 1.1579208923731619542e77,
                    1.3407807929942597099e154,
        };

        /*
         * Strip off leading blanks and check for a sign.
         */
        p = string;
        while (*p==' ' || *p=='\t')
                ++p;
        if (*p == '-') {
                sign = 1;
                ++p;
        } else if (*p == '+')
                ++p;

        /*
         * Count the number of digits in the mantissa (including the decimal
         * point), and also locate the decimal point.
         */
        decPt = -1;
        for (mantSize=0; ; ++mantSize) {
                c = *p;
                if (!isdigitx (c, radix)) {
                        if (c != '.' || decPt >= 0)
                                break;
                        decPt = mantSize;
                }
                ++p;
        }

        /*
         * Now suck up the digits in the mantissa.  Use two integers to
         * collect 9 digits each (this is faster than using floating-point).
         * If the mantissa has more than 18 digits, ignore the extras, since
         * they can't affect the value anyway.
         */
        pExp = p;
        p -= mantSize;
        if (decPt < 0)
                decPt = mantSize;
        else
                --mantSize;             /* One of the digits was the point. */

        switch (radix) {
        default:
        case 10: fracSz = 9;  fracOff = 1000000000; powTab = powersOf10; break;
        case 2:  fracSz = 30; fracOff = 1073741824; powTab = powersOf2;  break;
        case 8:  fracSz = 10; fracOff = 1073741824; powTab = powersOf8;  break;
        case 16: fracSz = 7;  fracOff = 268435456;  powTab = powersOf16; break;
        }
        if (mantSize > 2 * fracSz)
                mantSize = 2 * fracSz;
        fracExp = decPt - mantSize;
        if (mantSize == 0) {
                fraction = 0.0;
                p = string;
                goto done;
        } else {
                int frac1, frac2;

                for (frac1=0; mantSize>fracSz; --mantSize) {
                        c = *p++;
                        if (c == '.')
                                c = *p++;
                        frac1 = frac1 * radix + (c - '0');
                }
                for (frac2=0; mantSize>0; --mantSize) {
                        c = *p++;
                        if (c == '.')
                                c = *p++;
                        frac2 = frac2 * radix + (c - '0');
                }
                fraction = (double) fracOff * frac1 + frac2;
        }

        /*
         * Skim off the exponent.
         */
        p = pExp;
        if (*p=='E' || *p=='e' || *p=='S' || *p=='s' || *p=='F' || *p=='f' ||
             *p=='D' || *p=='d' || *p=='L' || *p=='l') {
                ++p;
                if (*p == '-') {
                        expSign = 1;
                        ++p;
                } else if (*p == '+')
                        ++p;
                while (isdigitx (*p, radix))
                        exp = exp * radix + (*p++ - '0');
        }
        if (expSign)
                exp = fracExp - exp;
        else
                exp = fracExp + exp;

        /*
         * Generate a floating-point number that represents the exponent.
         * Do this by processing the exponent one bit at a time to combine
         * many powers of 2 of 10. Then combine the exponent with the
         * fraction.
         */
        if (exp < 0) {
                expSign = 1;
                exp = -exp;
        } else
                expSign = 0;
        if (exp > maxExponent)
                exp = maxExponent;
        dblExp = 1.0;
        for (i=0; exp; exp>>=1, ++i)
                if (exp & 01)
                        dblExp *= powTab[i];
        if (expSign)
                fraction /= dblExp;
        else
                fraction *= dblExp;

```

```
210 done:
211         if (endPtr)
212                 *endPtr = p;
213 
214         return sign ? -fraction : fraction;
215 }
216 
217 #define BUFSIZE 7
218 int main()
219 {
220         char buf[BUFSIZE];
221         klee_make_symbolic (buf, sizeof buf, "buf");
222         klee_assume(buf[BUFSIZE-1]==0);
223 
224         // without passing return value out of main(), KLEE wouldn't work correctly
225         // perhaps, this is how we show that the return value is used somehow
226         // instead of dangling as void
227         return strtodx (buf, NULL, 10);
228 };
```

(https://smt.st/current_tree/KLEE/strtodx.c)

Interestingly, KLEE cannot handle floating-point arithmetic, but nevertheless, found something [4]:

```
% clang -emit-llvm -c -g -O0 -Xclang -disable-O0-optnone -I /tmp/klee_src/include/
   klee/ strtodx.c

% time klee --libc=uclibc --optimize strtodx.bc

...

KLEE: ERROR: strtodx.c:204: memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 303616
KLEE: done: completed paths = 2150
KLEE: done: partially completed paths = 1904
KLEE: done: generated tests = 2151

real    3m23.710s
user    2m48.737s

...

% ls klee-last | grep err
test000155.ptr.err

% ktest-tool klee-last/test000155.ktest | grep data
object 0: data: b'.0f-79\x00'
```

As it seems, the “.0f-79” string does out of bounds array access (read) at the line 204. During further investigation, I’ve found that `powersOf10[]` array is too short: 6th element (started at 0th) has been accessed. And here we see a part of array commented (line 81)! Probably someone’s mistake? Unfixed bug?

18.8 Simple expression evaluator (calculator)

I has been looking for simple expression evaluator (calculator in other words) which takes expression like “2+2” on input and gives answer. I’ve found one at http://stackoverflow.com/a/13895198.

```
// copypasted from http://stackoverflow.com/a/13895198 and reworked

```

[4] All timings are when KLEE 3.0 running on AMD Ryzen 5 3600.

```
// Bare bones scanner and parser for the following LL(1) grammar:
// expr -> term { [+-] term }     ; An expression is terms separated by add ops.
// term -> factor { [*/] factor } ; A term is factors separated by mul ops.
// factor -> unsigned_factor      ; A signed factor is a factor,
//         | - unsigned_factor    ;   possibly with leading minus sign
// unsigned_factor -> ( expr )    ; An unsigned factor is a parenthesized expression
//         | NUMBER               ;   or a number
//
// The parser returns the floating point value of the expression.

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>
#include <ctype.h>

#include "klee.h"

#define BUFSIZE 4
char input[BUFSIZE];
int input_idx=0;

char my_getchar()
{
        char rt=input[input_idx];
        input_idx++;
        return rt;
};

// The token buffer. We never check for overflow! Don't so in production code.
char buf[BUFSIZE];
int n = 0;

// The current character.
int ch;

// The look-ahead token.  This is the 1 in LL(1).
enum { ADD_OP, MUL_OP, LEFT_PAREN, RIGHT_PAREN, NOT_OP, NUMBER, END_INPUT } look_ahead;

// Forward declarations.
void init(void);
void advance(void);
int expr(void);
void error(char *msg);

// Parse expressions, one per line.
int main(void)
{
        // take input expression from input[]
        klee_make_symbolic(input, sizeof input, "input");
        klee_assume(input[BUFSIZE-1]==0);

        init();
        while (1)
        {
                int val = expr();
                //printf("%d\n", val);
                // without passing return value out of main(), KLEE wouldn't work correctly
                // perhaps, this is how we show that the return value is used somehow
                // instead of dangling as void
                return val;

                if (look_ahead != END_INPUT)
                        error("junk after expression");
                advance();  // past end of input mark
        }
        return 0;
}

// Just die on any error.
void error(char *msg)
{
        //fprintf(stderr, "Error: %s. Exiting.\n", msg);
        exit(1);
}

// Buffer the current character and read a new one.
void read()
{
        buf[n++] = ch;
        buf[n] = '\0';  // Terminate the string.
        ch = my_getchar();
}

// Ignore the current character.
void ignore()
{
        ch = my_getchar();
}

// Reset the token buffer.
void reset()
{
        n = 0;
        buf[0] = '\0';
}

// The scanner.  A tiny deterministic finite automaton.
int scan()
{
        reset();
START:
        // first character is digit?
        if (isdigit (ch))
                goto DIGITS;

        switch (ch)
        {
                case ' ': case '\t': case '\r':
                        ignore();
                        goto START;

                case '-': case '+': case '^':
                        read();
                        return ADD_OP;

                case '~':
                        read();
                        return NOT_OP;

                case '*': case '/': case '%':
                        read();
                        return MUL_OP;

                case '(':
                        read();
                        return LEFT_PAREN;

                case ')':
                        read();
                        return RIGHT_PAREN;

                case 0:
                case '\n':
                        ch = ' ';    // delayed ignore()
                        return END_INPUT;

                default:
                        //printf ("bad character: 0x%x\n", ch);
                        exit(0);
        }

DIGITS:
        if (isdigit (ch))
        {
                read();
                goto DIGITS;
        }
        else
                return NUMBER;
}

// To advance is just to replace the look-ahead.
void advance()
{
        look_ahead = scan();
}

// Clear the token buffer and read the first look-ahead.
void init()
{
        reset();
        ignore(); // junk current character
        advance();
}

int get_number(char *buf)
{
        char *endptr;

        int rt=strtoul (buf, &endptr, 10);

        // is the whole buffer has been processed?
        if (strlen(buf)!=endptr-buf)
        {
                //fprintf (stderr, "invalid number: %s\n", buf);
                exit(0);
        };
        return rt;
};
```

```c

int unsigned_factor()
{
        int rtn = 0;
        switch (look_ahead)
        {
                case NUMBER:
                        rtn=get_number(buf);
                        advance();
                        break;

                case LEFT_PAREN:
                        advance();
                        rtn = expr();
                        if (look_ahead != RIGHT_PAREN) error("missing ')'");
                        advance();
                        break;

                default:
                        //printf("unexpected token: %d\n", look_ahead);
                        exit(0);
        }
        return rtn;
}

int factor()
{
        int rtn = 0;
        // If there is a leading minus...
        if (look_ahead == ADD_OP && buf[0] == '-')
        {
                advance();
                rtn = -unsigned_factor();
        }
        else
                rtn = unsigned_factor();

        return rtn;
}

int term()
{
        int rtn = factor();
        while (look_ahead == MUL_OP)
        {
                switch(buf[0])
                {
                        case '*':
                                advance();
                                rtn *= factor();
                                break;

                        case '/':
                                advance();
                                rtn /= factor();
                                break;
                        case '%':
                                advance();
                                rtn %= factor();
                                break;
                }
245         }
246         return rtn;
247 }
248
249 int expr()
250 {
251         int rtn = term();
252         while (look_ahead == ADD_OP)
253         {
254                 switch(buf[0])
255                 {
256                         case '+':
257                                 advance();
258                                 rtn += term();
259                                 break;
260
261                         case '-':
262                                 advance();
263                                 rtn -= term();
264                                 break;
265                 }
266         }
267         return rtn;
268 }
```

(https://smt.st/current_tree/KLEE/calc.c)

I reworked it by commenting (f)printf() calls. Not needed anyway.

KLEE found two expression strings which leads to division error:

```
% clang -emit-llvm -c -g -O0 -Xclang -disable-O0-optnone -I /tmp/klee_src/include/
   klee/ calc.c

% time klee --libc=uclibc --optimize calc.bc

...

KLEE: ERROR: calc.c:235: divide by zero
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: calc.c:239: divide by zero
KLEE: NOTE: now ignoring this error at this location

(I had to interrupt it...)

KLEE: ctrl-c detected, requesting interpreter to halt.
KLEE: halting execution, dumping remaining states

KLEE: done: total instructions = 1629596851
KLEE: done: completed paths = 216
KLEE: done: partially completed paths = 8
KLEE: done: generated tests = 224

real    78m36.891s
user    78m22.369s
...

% ls klee-last | grep err
test000139.div.err
test000158.div.err

% ktest-tool klee-last/test000139.ktest | grep data
object 0: data: b'5/0\x00'
```

```
% ktest-tool klee-last/test000158.ktest | grep data
object 0: data: b'5%0\x00'
```

Maybe this is not an impressive result, nevertheless, it's yet another reminder that division and remainder operations must be wrapped somehow in production code to avoid possible crash.

18.9 More examples

https://feliam.wordpress.com/2010/10/07/the-symbolic-maze/

18.10 Exercise

Here is my crackme/keygenme, which may be tricky, but easy to solve using KLEE: http://challenges.re/74/.

Chapter 19

(Amateur) cryptography

19.1 *Professional* cryptography

Let's back to the method we previously used (17.1) to construct expressions using running Python function.

We can try to build expression for the output of XXTEA encryption algorithm:

```
#!/usr/bin/env python3

class Expr:
    def __init__(self,s):
        self.s=s

    def __str__(self):
        return self.s

    def convert_to_Expr_if_int(self, n):
        if isinstance(n, int):
            return Expr(str(n))
        if isinstance(n, Expr):
            return n
        raise AssertionError # unsupported type

    def __xor__(self, other):
        return Expr("(" + self.s + "^" + self.convert_to_Expr_if_int(other).s + ")")

    def __mul__(self, other):
        return Expr("(" + self.s + "*" + self.convert_to_Expr_if_int(other).s + ")")

    def __add__(self, other):
        return Expr("(" + self.s + "+" + self.convert_to_Expr_if_int(other).s + ")")

    def __and__(self, other):
        return Expr("(" + self.s + "&" + self.convert_to_Expr_if_int(other).s + ")")

    def __lshift__(self, other):
        return Expr("(" + self.s + "<<" + self.convert_to_Expr_if_int(other).s + ")")

    def __rshift__(self, other):
        return Expr("(" + self.s + ">>" + self.convert_to_Expr_if_int(other).s + ")")

    def __getitem__(self, d):
        return Expr("(" + self.s + "[" + d.s + "])")

# reworked from:

# Pure Python (2.x) implementation of the XXTEA cipher
# (c) 2009. Ivan Voras <ivoras@gmail.com>
# Released under the BSD License.

def raw_xxtea(v, n, k):

    def MX():
        return ((z>>5)^(y<<2)) + ((y>>3)^(z<<4))^(sum^y) + (k[(Expr(str(p)) & 3)^e]^z
            )

    y = v[0]
    sum = Expr("0")
    DELTA = 0x9e3779b9
    # Encoding only
    z = v[n-1]

    # number of rounds:
    #q = 6 + 52 / n
    q=1

    while q > 0:
        q -= 1
        sum = sum + DELTA
        e = (sum >> 2) & 3
        p = 0
        while p < n - 1:
            y = v[p+1]
            z = v[p] = v[p] + MX()
            p += 1
        y = v[0]
        z = v[n-1] = v[n-1] + MX()
    return 0

v=[Expr("input1"), Expr("input2"), Expr("input3"), Expr("input4")]
k=Expr("key")

raw_xxtea(v, 4, k)

for i in range(4):
    print (i, ":", v[i])
#print len(str(v[0]))+len(str(v[1]))+len(str(v[2]))+len(str(v[3]))
```

A key is chosen according to input data, and, obviously, we can't know it during symbolic execution, so we leave expression like `k[...]`.

Now results for just one round, for each of 4 outputs:

```
0 : (input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))^(((0+2654435769)^
    input2)+
((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))

1 : (input2+(((((input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))
    ^(((0+2654435769)^
input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))>>5)^(input3<<2))+((input3
    >>3)^((input1+
((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))^(((0+2654435769)^input2)+((key
    [((0&3)^
(((0+2654435769)>>2)&3))])^input4))))<<4)))^(((0+2654435769)^input3)+((key[((1&3)
    ^(((0+2654435769)>>2)&
3))])^(input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))^(((0+2654435769)
    ^input2)+
((key[((0&3)^(((0+2654435769)>>2)&3))])^input4)))))))

2 : (input3+(((((input2+(((((input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4
    <<4)))^
```

```
(((0+2654435769)^input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))>>5)^(
    input3<<2))+
((input3>>3)^((input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))
    ^(((0+2654435769)^
input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))<<4)))^(((0+2654435769)^
    input3)+
((key[((1&3)^(((0+2654435769)>>2)&3))])^(input1+((((input4>>5)^(input2<<2))+((input2
    >>3)^(input4<<4)))
^(((0+2654435769)^input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))))))>>5)
    ^(input4<<2))+
((input4>>3)^((input2+(((((input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4
    <<4)))^(((0+2654435769)^
input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))>>5)^(input3<<2))+((input3
    >>3)^((input1+
((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))^(((0+2654435769)^input2)+((key
    [((0&3)^
(((0+2654435769)>>2)&3))])^input4))))<<4)))^(((0+2654435769)^input3)+((key[((1&3)
    ^(((0+2654435769)>>2)&3))])^
(input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))^(((0+2654435769)^
    input2)+((key[((0&3)^
(((0+2654435769)>>2)&3))])^input4))))))))<<4)))^(((0+2654435769)^input4)+((key[((2&3)
    ^(((0+2654435769)>>2)&
3))])^(input2+(((((input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))
    ^(((0+2654435769)^
input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))>>5)^(input3<<2))+((input3
    >>3)^((input1+
((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))^(((0+2654435769)^input2)+((key
    [((0&3)^
(((0+2654435769)>>2)&3))])^input4))))<<4)))^(((0+2654435769)^input3)+((key[((1&3)
    ^(((0+2654435769)>>2)&
3))])^(input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))^(((0+2654435769)
    ^input2)+((key[((0&3)^
(((0+2654435769)>>2)&3))])^input4)))))))))))

3 : (input4+(((((input3+(((((input2+(((((input1+((((input4>>5)^(input2<<2))+((input2
    >>3)^(input4<<4)))^
(((0+2654435769)^input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))>>5)^(
    input3<<2))+((input3>>3)^
((input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))^(((0+2654435769)^
    input2)+((key[((0&3)^
(((0+2654435769)>>2)&3))])^input4))))<<4)))^(((0+2654435769)^input3)+((key[((1&3)
    ^(((0+2654435769)>>2)&3))])^
(input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))^(((0+2654435769)^
    input2)+((key[((0&3)^
(((0+2654435769)>>2)&3))])^input4))))))))>>5)^(input4<<2))+((input4>>3)^((input2
    +(((((input1+((((input4>>5)^
(input2<<2))+((input2>>3)^(input4<<4)))^(((0+2654435769)^input2)+((key[((0&3)
    ^(((0+2654435769)>>2)&3))])^
input4))))>>5)^(input3<<2))+((input3>>3)^((input1+((((input4>>5)^(input2<<2))+((
    input2>>3)^(input4<<4)))^
(((0+2654435769)^input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))<<4)))
    ^(((0+2654435769)^input3)+
((key[((1&3)^(((0+2654435769)>>2)&3))])^(input1+((((input4>>5)^(input2<<2))+((input2
    >>3)^(input4<<4)))^
(((0+2654435769)^input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))))))<<4)))
    ^(((0+2654435769)^
input4)+((key[((2&3)^(((0+2654435769)>>2)&3))])^(input2+(((((input1+((((input4>>5)^(
    input2<<2))+((input2>>3)^
(input4<<4)))^(((0+2654435769)^input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4
    ))))>>5)^(input3<<2))+
((input3>>3)^((input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))
    ^(((0+2654435769)^input2)+
((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))<<4)))^(((0+2654435769)^input3)+((
    key[((1&3)^(((0+2654435769)>>
2)&3))])^(input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))
    ^(((0+2654435769)^input2)+((key[((0&3)^
(((0+2654435769)>>2)&3))])^input4)))))))))))>>5)^((input1+((((input4>>5)^(input2<<2)
    )+((input2>>3)^(input4<<
4)))^(((0+2654435769)^input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))<<2))
    +(((input1+((((input4>>5)^
(input2<<2))+((input2>>3)^(input4<<4)))^(((0+2654435769)^input2)+((key[((0&3)
    ^(((0+2654435769)>>2)&3))])^
input4))))>>3)^((input3+(((((input2+(((((input1+((((input4>>5)^(input2<<2))+((input2
    >>3)^(input4<<4)))^(((0+
2654435769)^input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))>>5)^(input3
    <<2))+((input3>>3)^((input1+
((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))^(((0+2654435769)^input2)+((key
    [((0&3)^(((0+2654435769)>>
2)&3))])^input4))))<<4)))^(((0+2654435769)^input3)+((key[((1&3)^(((0+2654435769)>>2)
    &3))])^(input1+((((input4>>
5)^(input2<<2))+((input2>>3)^(input4<<4)))^(((0+2654435769)^input2)+((key[((0&3)
    ^(((0+2654435769)>>2)&3))])^
input4))))))))>>5)^(input4<<2))+((input4>>3)^((input2+(((((input1+((((input4>>5)^(
    input2<<2))+((input2>>3)^
(input4<<4)))^(((0+2654435769)^input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4
    ))))>>5)^(input3<<2))+
((input3>>3)^((input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))
    ^(((0+2654435769)^input2)+
((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))<<4)))^(((0+2654435769)^input3)+((
    key[((1&3)^(((0+2654435769)>>
2)&3))])^(input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))
    ^(((0+2654435769)^input2)+((key[((0&3)^
(((0+2654435769)>>2)&3))])^input4))))))))<<4)))^(((0+2654435769)^input4)+((key[((2&3)
    ^(((0+2654435769)>>2)&3))])^
(input2+(((((input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))
    ^(((0+2654435769)^input2)+((key[((0&3)^
(((0+2654435769)>>2)&3))])^input4))))>>5)^(input3<<2))+((input3>>3)^((input1+((((
    input4>>5)^(input2<<2))+
((input2>>3)^(input4<<4)))^(((0+2654435769)^input2)+((key[((0&3)^(((0+2654435769)>>2)
    &3))])^input4))))<<4)))^(((
0+2654435769)^input3)+((key[((1&3)^(((0+2654435769)>>2)&3))])^(input1+((((input4>>5)
    ^(input2<<2))+((input2>>3)^(
input4<<4)))^(((0+2654435769)^input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4)
    ))))))))))<<4)))^(((0+
2654435769)^(input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))
    ^(((0+2654435769)^input2)+((key[((0&3)^
(((0+2654435769)>>2)&3))])^input4)))))+((key[((3&3)^(((0+2654435769)>>2)&3))])^(
    input3+(((((input2+(((((input1+
((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))^(((0+2654435769)^input2)+((key
    [((0&3)^(((0+2654435769)>>
2)&3))])^input4))))>>5)^(input3<<2))+((input3>>3)^((input1+((((input4>>5)^(input2<<2)
    )+((input2>>3)^(input4<<
4)))^(((0+2654435769)^input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))<<4))
    )^(((0+2654435769)^input3)+
((key[((1&3)^(((0+2654435769)>>2)&3))])^(input1+((((input4>>5)^(input2<<2))+((input2
    >>3)^(input4<<4)))^(((0+
2654435769)^input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))))))>>5)^(
    input4<<2))+((input4>>3)^((
input2+(((((input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))
    ^(((0+2654435769)^input2)+((key[((0&3)^
(((0+2654435769)>>2)&3))])^input4))))>>5)^(input3<<2))+((input3>>3)^((input1+((((
    input4>>5)^(input2<<2))+((
input2>>3)^(input4<<4)))^(((0+2654435769)^input2)+((key[((0&3)^(((0+2654435769)>>2)
    &3))])^input4))))<<4)))^(((0+
2654435769)^input3)+((key[((1&3)^(((0+2654435769)>>2)&3))])^(input1+((((input4>>5)^(
    input2<<2))+((input2>>3)^
(input4<<4)))^(((0+2654435769)^input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4
    ))))))))<<4)))^(((0+
2654435769)^input4)+((key[((2&3)^(((0+2654435769)>>2)&3))])^(input2+(((((input1+((((
    input4>>5)^(input2<<2))+
((input2>>3)^(input4<<4)))^(((0+2654435769)^input2)+((key[((0&3)^(((0+2654435769)>>2)
    &3))])^input4))))>>5)^
(input3<<2))+((input3>>3)^((input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4
    <<4)))^(((0+2654435769)^
input2)+((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))<<4)))^(((0+2654435769)^
    input3)+((key[((1&3)^(((0+
2654435769)>>2)&3))])^(input1+((((input4>>5)^(input2<<2))+((input2>>3)^(input4<<4)))
    ^(((0+2654435769)^input2)+
((key[((0&3)^(((0+2654435769)>>2)&3))])^input4))))))))))))))
```

Somehow, the size of the expression for each subsequent output is bigger. I hope I haven't been mistaken? And this is just for 1 round. For 2 rounds, size of all 4 expression is ≈ $970KB$. For 3 rounds, this is ≈ $115MB$. For 4 rounds, I have not enough RAM on my computer. Expressions *exploding* exponentially. And there are 19 rounds. You can weigh it.

Perhaps, you can simplify these expressions: there are a lot of excessive parenthesis, but I'm highly pessimistic, cryptoalgorithms constructed in such a way to not have any spare operations.

In order to crack it, you can use these expressions as system of equation and try to solve it using SMT-solver. This is called "algebraic attack".

In other words, theoretically, you can build a system of equation like this: $MD5(x) = 12341234...$, but expressions are so huge so it's impossible to solve them. Yes, cryptographers are fully aware of this and one of the goals of the successful cipher is to make expressions as big as possible, using reasonable time and size of algorithm.

Nevertheless, you can find numerous papers about breaking these cryptosystems with reduced number of rounds: when expression isn't *exploded* yet, sometimes it's possible. This cannot be applied in practice, but such an experience has some interesting theoretical uses.

19.1.1 Attempts to break "serious" crypto

CryptoMiniSat itself exist to support XOR operation, which is ubiquitous in cryptography.

- Bitcoin mining with CBMC and SAT solver: `http://jheusser.github.io/2013/02/03/satcoin.html`, `https://github.com/msoos/sha256-sat-bitcoin`.
- Alexander Semenov, attempts to break A5/1, etc. (Russian presentation)
- Vegard Nossum - SAT-based preimage attacks on SHA-1
- Algebraic Attacks on the Crypto-1 Stream Cipher in MiFare Classic and Oyster Cards
- Attacking Bivium Using SAT Solvers
- Extending SAT Solvers to Cryptographic Problems
- Applications of SAT Solvers to Cryptanalysis of Hash Functions
- Algebraic-Differential Cryptanalysis of DES
- Attempt to break Speck cipher: `https://github.com/TechSecCTF/z3_splash_class`.

19.2 Amateur cryptography

(Another term is *Low-Complexity Cryptography*.)

This is what you can find in serial numbers, license keys, executable file packers, CTF[1], malware, etc. Sometimes even ransomware (but rarely nowadays, in 2017-2021).

Java obfuscators encrypts text strings using very simple algorithms.

[1] Capture the Flag

Amateur cryptography is often can be broken using SMT solver, or even KLEE.

Amateur cryptography is usually based not on theory, but on visual complexity: if its creator getting results which are seems chaotic enough, often, one stops to improve it further. This is security based not even on obscurity, but on a chaotic mess. This is also sometimes called "The Fallacy of Complex Manipulation" [see RFC4086].

Devising your own cryptoalgorithm is a very tricky thing to do. This can be compared to devising your own PRNG. Even famous Donald Knuth in 1959 constructed one, and it was visually very complex, but, as it turns out in practice, it has very short cycle of length 3178 [see TAOCP vol.II page 4, (1997).]

The very first problem is that making an algorithm which can generate very long expressions is tricky thing itself. A common mistake is to use operations like XOR and rotations/permutations, which can't help much. Even worse: some people think that XORing a value several times can be better, like: $(x \oplus 1234) \oplus 5678$. Obviously, these two XOR operations (or more precisely, any number of it) can be reduced to a single one. Same story about applied operations like addition and subtraction—they all also can be reduced to single one.

Real cryptoalgorithms, like IDEA, can use several operations from different groups, like XOR, addition and multiplication. Applying them all in specific order will make resulting expression irreducible.

When I prepared this article, I tried to make an example of such amateur hash function:

```
// copypasted from http://blog.regehr.org/archives/1063
uint32_t rotl32b (uint32_t x, uint32_t n)
{
                assert (n<32);
                        if (!n) return x;
                                return (x<<n) | (x>>(32-n));
}

uint32_t rotr32b (uint32_t x, uint32_t n)
{
                assert (n<32);
                        if (!n) return x;
                                return (x>>n) | (x<<(32-n));
}

void megahash (uint32_t buf[4])
{
        for (int i=0; i<4; i++)
        {
                uint32_t t0=buf[0]^0x12345678^buf[1];
                uint32_t t1=buf[1]^0xabcdef01^buf[2];
                uint32_t t2=buf[2]^0x23456789^buf[3];
                uint32_t t3=buf[3]^0x0abcdef0^buf[0];

                buf[0]=rotl32b(t0, 1);
                buf[1]=rotr32b(t1, 2);
                buf[2]=rotl32b(t2, 3);
                buf[3]=rotr32b(t3, 4);
        };
};

int main()
{
        uint32_t buf[4];
        klee_make_symbolic(buf, sizeof buf);
        megahash (buf);
        if (buf[0]==0x18f71ce6          // or whatever
                && buf[1]==0xf37c2fc9
                && buf[2]==0x1cfe96fe
                && buf[3]==0x8c02c75e)
                klee_assert(0);
};
```

KLEE can break it with little effort. Functions of such complexity is common in shareware, which checks license keys, etc.

Here is how we can make its work harder by making rotations dependent of inputs, and this makes number of possible inputs much, much bigger:

```
void megahash (uint32_t buf[4])
{
        for (int i=0; i<16; i++)
        {
                uint32_t t0=buf[0]^0x12345678^buf[1];
                uint32_t t1=buf[1]^0xabcdef01^buf[2];
                uint32_t t2=buf[2]^0x23456789^buf[3];
                uint32_t t3=buf[3]^0x0abcdef0^buf[0];

                buf[0]=rotl32b(t0, t1&0x1F);
                buf[1]=rotr32b(t1, t2&0x1F);
                buf[2]=rotl32b(t2, t3&0x1F);
                buf[3]=rotr32b(t3, t0&0x1F);
        };
};
```

Addition (or *modular addition*[2], as cryptographers say) can make things even harder:

```
void megahash (uint32_t buf[4])
{
        for (int i=0; i<4; i++)
        {
                uint32_t t0=buf[0]^0x12345678^buf[1];
                uint32_t t1=buf[1]^0xabcdef01^buf[2];
                uint32_t t2=buf[2]^0x23456789^buf[3];
                uint32_t t3=buf[3]^0x0abcdef0^buf[0];

                buf[0]=rotl32b(t0, t2&0x1F)+t1;
                buf[1]=rotr32b(t1, t3&0x1F)+t2;
                buf[2]=rotl32b(t2, t1&0x1F)+t3;
                buf[3]=rotr32b(t3, t2&0x1F)+t4;
        };
};
```

Heavy operations for SAT/SMT are shifts/rotates by a variable, division, remainder. *Easy operations*: shifts/rotates by constant, bit twiddling.

As en exercise, you can try to make a block cipher which KLEE wouldn't break. This is quite sobering experience.

Another exercise is to make an algorithm that converts a plain text in English or your native language to the block of the same length, but with highest possible entropy. It's not an easy task. (Data compression is cheating.)

An easy way to test your algorithm: encrypt numbers starting at 0 and feed the resulting ciphertext to *diehard test* [4] (like in Counter/CTR encryption mode). These tests are designed to check PRNGs. In other words, these tests shouldn't find any regularities in a list of random numbers, as well as in a ciphertext.

Summary: if you deal with amateur cryptography, you can always give KLEE and SMT solver a try. Even more: sometimes you have only decryption function, and if algorithm is simple enough, KLEE or SMT solver can reverse things back.

If a SAT/SMT solver can find a key faster than bruteforce, this is usually a very bad symptom for the algorithm.

One amusing thing to mention: if you try to implement amateur cryptoalgorithm in Verilog/VHDL language to run it on FPGA, maybe in brute-force way, you can find that EDA[5] tools can optimize many things during synthesis (this is the word they use for "compilation") and can leave cryptoalgorithm much smaller/simpler than it was. Even if you try to implement DES[6] algorithm *in bare metal* with a fixed key, Altera Quartus can optimize first round of DES and make it slightly smaller/faster than others.

19.2.1 Avalanche effect

A significant property of the *serious* cryptography is: "The two inputs differ by only a single bit, but approximately half the bits are different in the digests." [cited from Alan A. A. Donovan, Brian W. Kernighan — The Go Programming

[2] Read more about it in Untitled collection of math notes[3]

[4] https://en.wikipedia.org/wiki/Diehard_tests

[5] Electronic design automation

[6] Data Encryption Standard

Language]. This is also known as the *avalanche effect* in cryptography.

Let's see, how two SHA256 hashes will differ if the input strings are differ in one bit?

Listing 19.1: *NIX shell

```
% echo hello0 | sha256sum
aa36f6c221b2295332d8de275e4b9e1fd2383a8ca3f2598022958b82a1f85ce0  -

% echo hello1 | sha256sum
8d678c54ff53cac1efd6d62697aeb5a005491cd04b83228fd8411ba9f02b98e7  -
```

Listing 19.2: Wolfram Mathematica

```
In[]:= a=16^^aa36f6c221b2295332d8de275e4b9e1fd2383a8ca3f2598022958b82a1f85ce0
Out[]= 76990297064032879667747642138410318794708739067059518462895617345038449138912

In[]:= b=16^^8d678c54ff53cac1efd6d62697aeb5a005491cd04b83228fd8411ba9f02b98e7
Out[]= 63959065433925909480062682925690708784612992082320075146058498580507093866727

In[]:= diff=BitXor[a,b]
Out[]= 17784161787513000877216087191171407793188241648243295249170496887068053193735

In[]:= DigitCount[diff, 2, 1]
Out[]= 130

In[]:= 256/130//N
Out[]= 1.96923
```

Almost a half (130 bits of 256).

19.2.2 Bugs

Another prominent feature of amateur cryptography is bugs. Bugs here often left uncaught because output of encrypting function visually looked "good enough" or "obfuscated enough", so a developer stopped to work on it.

This is especially feature of hash functions, because when you work on block cipher, you have to do two functions (encryption/decryption), while hashing function is single.

Weirdest ever amateur encryption algorithm I once saw, encrypted only odd bytes of input block, while even bytes left untouched, so the input plain text has been partially seen in the resulting encrypted block. It was encryption routine used in license key validation. Hard to believe someone did this on purpose. Most likely, it was just an unnoticed bug.

Not uncommon thing is when both encryption and decryption functions are laying side-by-side in the code: probably a programmer just wrote two functions and they are residing in the same source code file/object file. So if you see a license decryption function in code, try to inspect functions near it.

Also, not uncommon bug is when a encryption key(s) are present in the executable file. Even, (RSA) private keys that can be used to sign license.

19.2.3 XOR ciphers

Simplest possible amateur cryptography is just application of XOR operation using some kind of table. Maybe even simpler. This is a real algorithm I once saw:

```
for (i=0; i<size; i++)
    buf[i]=buf[i]^(31*(i+1));
```

This is not even encryption, rather concealing or hiding.

Some other examples of simple XOR-cipher cryptoanalysis are present in the "Reverse Engineering for Beginners" [7] book.

19.2.4 Other features

Tables There are often table(s) with pseudorandom data, which is/are used chaotically.

[7] http://beginners.re

Checksumming End-users can have proclivity of changing license codes, serial numbers, etc., with a hope this could affect behaviour of software. So there is often some kind of checksum: starting at simple summing and CRC. This is close to MAC[8] in real cryptography.

Entropy level Maybe (much) lower, despite the fact that data looks random.

19.2.5 Examples of amateur cryptography

- A popular FLEXlm license manager was based on a simple amateur cryptoalgorithm (before they switched to ECC[9]), which can be cracked easily.
- Pegasus Mail Password Decoder: http://phrack.org/issues/52/3.html - a very typical example.
- You can find a lot of blog posts about breaking CTF-level crypto using Z3, etc. Like: https://yurichev.com/mirrors/SAT_SMT_crypto/Keygenning%20using%20the%20Z3%20SMT%20Solver.pdf, http://doar-e.github.io/blog/2015/08/18/keygenning-with-klee/.
- Many examples of amateur cryptography I've taken from an old Fravia website, etc: https://yurichev.com/mirrors/amateur_crypto_examples_from_Fravia/, https://yurichev.com/mirrors/amateur_crypto/aol_aim.txt.
- Dmitry Sklyarov's book "The art of protecting and breaking information" (2004, in Russian) contains many examples of amateur cryptography and misused cryptographical algorithms as well.

This is a real snippet from some software. This function converts MAC address of your network card or HDD's serial number to a license key that is to be compared with what user is supplied.

Note usage of srand()/rand() functions as blackbox functions.

```
#include <stdio.h>

int __cdecl sub_832EC0(unsigned int MAC_or_HDD_SN, char* out_str)
{
  unsigned int v4; // eax
  signed int v5; // ST24_4
  int v6; // edi
  int v7; // edi
  int v8; // eax
  int v2; // [esp+8h] [ebp+8h]
  int v1; // [esp+Ch] [ebp+Ch]

  v4 = MAC_or_HDD_SN;
  if ( MAC_or_HDD_SN <= 1000000 )
  {
    if ( MAC_or_HDD_SN <= 1000 )
      v4 = 1001001 * MAC_or_HDD_SN;
    else
      v4 = 1001 * MAC_or_HDD_SN;
  }
  v5 = 112 + v4 % (532123 + 916245078);
  v6 = (signed int)(112 + v4 % (532123 + 916245078)) % 1000000;
  v2 = v5 % 1000;
  srand(v5);
  v1 = rand() % 1000;
  srand(v6);
  v7 = rand() % 1000;
  srand(v2);
  v8 = rand();
  sprintf(
           out_str,
           (const char *)L"%03d%03d%03d%03d",
           v1,
           v7,
```

[8] Message authentication code
[9] Elliptic curve cryptography

```
           v8 % 1000,
           (v8 % 1000 + 1000 * (v7 + 1000 * v1)) % 719);
}

int main()
{
        char tmp[128];
        sub_832EC0(0x29d989d4, tmp);
        printf ("%s\n", tmp);
};
```

19.2.6 Examples of breaking it using SAT/SMT solvers

- Reversing the petya ransomware with constraint solvers (Z3) [10].
- Dcoder – kao's "Toy Project" and Algebraic Cryptanalysis [11].

19.2.7 Case study: a simple hash function

(This piece of text was initially added to my "Reverse Engineering for Beginners" book (beginners.re) at March 2014) [12].

Here is one-way hash function, that converted a 64-bit value to another and we need to try to reverse its flow back.

Manual decompiling

Here its assembly language listing in IDA:

```
sub_401510      proc near
                ; ECX = input
                mov     rdx, 5D7E0D1F2E0F1F84h
                mov     rax, rcx        ; input
                imul    rax, rdx
                mov     rdx, 388D76AEE8CB1500h
                mov     ecx, eax
                and     ecx, 0Fh
                ror     rax, cl
                xor     rax, rdx
                mov     rdx, 0D2E9EE7E83C4285Bh
                mov     ecx, eax
                and     ecx, 0Fh
                rol     rax, cl
                lea     r8, [rax+rdx]
                mov     rdx, 8888888888888889h
                mov     rax, r8
                mul     rdx
                shr     rdx, 5
                mov     rax, rdx
                lea     rcx, [r8+rdx*4]
                shl     rax, 6
                sub     rcx, rax
                mov     rax, r8
                rol     rax, cl
                ; EAX = output
                retn
sub_401510      endp
```

The example was compiled by GCC, so the first argument is passed in ECX.

[10] https://0xec.blogspot.com/2016/04/reversing-petya-ransomware-with.html

[11] https://yurichev.com/mirrors/SAT_SMT_crypto/solution.pdf

[12] This example was also used by Murphy Berzish in his lecture about SAT and SMT: http://mirror.csclub.uwaterloo.ca/csclub/mtrberzi-sat-smt-slides.pdf, http://mirror.csclub.uwaterloo.ca/csclub/mtrberzi-sat-smt.mp4

If you don't have Hex-Rays or Ghidra, or if you don't trust to it, you can try to reverse this code manually. One method is to represent the CPU registers as local C variables and replace each instruction by a one-line equivalent expression, like:

```
uint64_t f(uint64_t input)
{
        uint64_t rax, rbx, rcx, rdx, r8;

        ecx=input;

        rdx=0x5D7E0D1F2E0F1F84;
        rax=rcx;
        rax*=rdx;
        rdx=0x388D76AEE8CB1500;
        rax=_lrotr(rax, rax&0xF); // rotate right
        rax^=rdx;
        rdx=0xD2E9EE7E83C4285B;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+rdx;
        rdx=0x8888888888888889;
        rax=r8;
        rax*=rdx;
        rdx=rdx>>5;
        rax=rdx;
        rcx=r8+rdx*4;
        rax=rax<<6;
        rcx=rcx-rax;
        rax=r8
        rax=_lrotl (rax, rcx&0xFF); // rotate left
        return rax;
};
```

If you are careful enough, this code can be compiled and will even work in the same way as the original.

Then, we are going to rewrite it gradually, keeping in mind all registers usage. Attention and focus is very important here—any tiny typo may ruin all your work!

Here is the first step:

```
uint64_t f(uint64_t input)
{
        uint64_t rax, rbx, rcx, rdx, r8;

        ecx=input;

        rdx=0x5D7E0D1F2E0F1F84;
        rax=rcx;
        rax*=rdx;
        rdx=0x388D76AEE8CB1500;
        rax=_lrotr(rax, rax&0xF); // rotate right
        rax^=rdx;
        rdx=0xD2E9EE7E83C4285B;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+rdx;

        rdx=0x8888888888888889;
        rax=r8;
        rax*=rdx;
        // RDX here is a high part of multiplication result
        rdx=rdx>>5;
        // RDX here is division result!
        rax=rdx;

        rcx=r8+rdx*4;
        rax=rax<<6;
        rcx=rcx-rax;
        rax=r8
        rax=_lrotl (rax, rcx&0xFF); // rotate left
        return rax;
};
```

Next step:

```
uint64_t f(uint64_t input)
{
        uint64_t rax, rbx, rcx, rdx, r8;

        ecx=input;

        rdx=0x5D7E0D1F2E0F1F84;
        rax=rcx;
        rax*=rdx;
        rdx=0x388D76AEE8CB1500;
        rax=_lrotr(rax, rax&0xF); // rotate right
        rax^=rdx;
        rdx=0xD2E9EE7E83C4285B;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+rdx;

        rdx=0x8888888888888889;
        rax=r8;
        rax*=rdx;
        // RDX here is a high part of multiplication result
        rdx=rdx>>5;
        // RDX here is division result!
        rax=rdx;

        rcx=(r8+rdx*4)-(rax<<6);
        rax=r8
        rax=_lrotl (rax, rcx&0xFF); // rotate left
        return rax;
};
```

We can spot the division using multiplication. Indeed, let's calculate the divider in Wolfram Mathematica:

Listing 19.3: Wolfram Mathematica

```
In[1]:=N[2^(64 + 5)/16^^8888888888888889]
Out[1]:=60.
```

We get this:

```
uint64_t f(uint64_t input)
{
        uint64_t rax, rbx, rcx, rdx, r8;

        ecx=input;

        rdx=0x5D7E0D1F2E0F1F84;
        rax=rcx;
        rax*=rdx;
        rdx=0x388D76AEE8CB1500;
        rax=_lrotr(rax, rax&0xF); // rotate right
        rax^=rdx;
        rdx=0xD2E9EE7E83C4285B;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+rdx;
        rax=rdx=r8/60;

        rcx=(r8+rax*4)-(rax*64);
        rax=r8
        rax=_lrotl (rax, rcx&0xFF); // rotate left
        return rax;
};
```

One more step:

```
uint64_t f(uint64_t input)
{
        uint64_t rax, rbx, rcx, rdx, r8;

        rax=input;
        rax*=0x5D7E0D1F2E0F1F84;
        rax=_lrotr(rax, rax&0xF); // rotate right
        rax^=0x388D76AEE8CB1500;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+0xD2E9EE7E83C4285B;

        rcx=r8-(r8/60)*60;
        rax=r8
        rax=_lrotl (rax, rcx&0xFF); // rotate left
        return rax;
};
```

By simple reducing, we finally see that it's calculating the remainder, not the quotient:

```
uint64_t f(uint64_t input)
{
        uint64_t rax, rbx, rcx, rdx, r8;

        rax=input;
        rax*=0x5D7E0D1F2E0F1F84;
        rax=_lrotr(rax, rax&0xF); // rotate right
        rax^=0x388D76AEE8CB1500;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+0xD2E9EE7E83C4285B;

        return _lrotl (r8, r8 % 60); // rotate left
};
```

We end up with this fancy formatted source-code:

```
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include <intrin.h>

#define C1 0x5D7E0D1F2E0F1F84
#define C2 0x388D76AEE8CB1500
#define C3 0xD2E9EE7E83C4285B

uint64_t hash(uint64_t v)
{
        v*=C1;
        v=_lrotr(v, v&0xF); // rotate right
        v^=C2;
        v=_lrotl(v, v&0xF); // rotate left
        v+=C3;
        v=_lrotl(v, v % 60); // rotate left
        return v;
};

int main()
{
        printf ("%llu\n", hash(...));
};
```

Since we are not cryptanalysts we can't find an easy way to generate the input value for some specific output value. The rotate instruction's coefficients look frightening—it's a warranty that the function is not bijective, it is rather surjective, it has collisions, or, speaking more simply, many inputs may be mapped to one output.

Brute-force is not solution because values are 64-bit ones, that's beyond reality.

Now let's use Z3

Still, without any special cryptographic knowledge, we may try to break this algorithm using Z3.

Here is the Python source code:

```
 1 #!/usr/bin/env python3
 2 
 3 from z3 import *
 4 
 5 C1=0x5D7E0D1F2E0F1F84
 6 C2=0x388D76AEE8CB1500
 7 C3=0xD2E9EE7E83C4285B
 8 
 9 inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i6 outp', 64)
10 
11 s = Solver()
12 s.add(i1==inp*C1)
13 s.add(i2==RotateRight (i1, i1 & 0xF))
14 s.add(i3==i2 ^ C2)
15 s.add(i4==RotateLeft(i3, i3 & 0xF))
16 s.add(i5==i4 + C3)
17 s.add(outp==RotateLeft (i5, URem(i5, 60)))
18 
19 s.add(outp==10816636949158156260)
20 
21 print (s.check())
22 m=s.model()
23 print (m)
24 print (" inp=0x%X" % m[inp].as_long())
25 print ("outp=0x%X" % m[outp].as_long())
```

This is going to be our first solver.

We see the variable definitions on line 7. These are just 64-bit variables. `i1..i6` are intermediate variables, representing the values in the registers between instruction executions.

Then we add the so-called constraints on lines 10..15. The last constraint at 17 is the most important one: we are going to try to find an input value for which our algorithm will produce 10816636949158156260.

RotateRight, RotateLeft, URem—are functions from Z3 Python API, not related to Python language.

Then we run it:

```
...>python.exe 1.py
sat
[i1 = 3959740824832824396,
 i3 = 8957124831728646493,
 i5 = 10816636949158156260,
 inp = 1364123924608584563,
 outp = 10816636949158156260,
 i4 = 14065440378185297801,
 i2 = 4954926323707358301]
 inp=0x12EE577B63E80B73
outp=0x961C69FF0AEFD7E4
```

"sat" mean "satisfiable", i.e., the solver was able to find at least one solution. The solution is printed in the square brackets. The last two lines are the input/output pair in hexadecimal form. Yes, indeed, if we run our function with 0x12EE577B63E80B73 as input, the algorithm will produce the value we were looking for.

But, as we noticed before, the function we work with is not bijective, so there may be other correct input values. An SMT solver is not capable of producing more than one result, but let's hack our example slightly, by adding line 21, which implies "look for any other results than this":

```
1  #!/usr/bin/env python3
2  
3  from z3 import *
4  
5  C1=0x5D7E0D1F2E0F1F84
6  C2=0x388D76AEE8CB1500
7  C3=0xD2E9EE7E83C4285B
8  
9  inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i6 outp', 64)
10 
11 s = Solver()
12 s.add(i1==inp*C1)
13 s.add(i2==RotateRight (i1, i1 & 0xF))
14 s.add(i3==i2 ^ C2)
15 s.add(i4==RotateLeft(i3, i3 & 0xF))
16 s.add(i5==i4 + C3)
17 s.add(outp==RotateLeft (i5, URem(i5, 60)))
18 
19 s.add(outp==10816636949158156260)
20 
21 s.add(inp!=0x12EE577B63E80B73)
22 
23 print (s.check())
24 m=s.model()
25 print (m)
26 print (" inp=0x%X" % m[inp].as_long())
27 print ("outp=0x%X" % m[outp].as_long())
```

Indeed, it finds another correct result:

```
...>python.exe 2.py
sat
[i1 = 3959740824832824396,
 i3 = 8957124831728646493,
 i5 = 10816636949158156260,
 inp = 10587495961463360371,
 outp = 10816636949158156260,
 i4 = 14065440378185297801,
 i2 = 4954926323707358301]
 inp=0x92EE577B63E80B73
outp=0x961C69FF0AEFD7E4
```

This can be automated. Each found result can be added as a constraint and then the next result will be searched for. Here is a slightly more sophisticated example:

```
#!/usr/bin/env python3

from z3 import *

C1=0x5D7E0D1F2E0F1F84
C2=0x388D76AEE8CB1500
C3=0xD2E9EE7E83C4285B

inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i6 outp', 64)

s = Solver()
```

```
12 s.add(i1==inp*C1)
13 s.add(i2==RotateRight (i1, i1 & 0xF))
14 s.add(i3==i2 ^ C2)
15 s.add(i4==RotateLeft(i3, i3 & 0xF))
16 s.add(i5==i4 + C3)
17 s.add(outp==RotateLeft (i5, URem(i5, 60)))
18
19 s.add(outp==10816636949158156260)
20
21 # copypasted from http://stackoverflow.com/questions/11867611/z3py-checking-all-
       solutions-for-equation
22 result=[]
23 while True:
24     if s.check() == sat:
25         m = s.model()
26         print (m[inp])
27         result.append(m)
28         # Create a new constraint the blocks the current model
29         block = []
30         for d in m:
31             # d is a declaration
32             if d.arity() > 0:
33                 raise Z3Exception("uninterpreted functions are not supported")
34             # create a constant from declaration
35             c=d()
36             if is_array(c) or c.sort().kind() == Z3_UNINTERPRETED_SORT:
37                 raise Z3Exception("arrays and uninterpreted sorts are not supported")
38             block.append(c != m[d])
39         s.add(Or(block))
40     else:
41             print ("results total=",len(result))
42             break
```

We got:

```
1364123924608584563
1234567890
9223372038089343698
4611686019661955794
13835058056516731602
3096040143925676201
12319412180780452009
7707726162353064105
16931098199207839913
1906652839273745429
11130024876128521237
15741710894555909141
6518338857701133333
5975809943035972467
15199181979890748275
10587495961463360371
results total= 16
```

So there are 16 correct input values for 0x92EE577B63E80B73 as a result.

The second is 1234567890—it is indeed the value which was used by me originally while preparing this example.

Let's also try to research our algorithm a bit more. Acting on a sadistic whim, let's find if there are any possible input/output pairs in which the lower 32-bit parts are equal to each other?

Let's remove the *outp* constraint and add another, at line 17:

```
1 #!/usr/bin/env python3
2
3 from z3 import *
4
```

```
C1=0x5D7E0D1F2E0F1F84
C2=0x388D76AEE8CB1500
C3=0xD2E9EE7E83C4285B

inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i6 outp', 64)

s = Solver()
s.add(i1==inp*C1)
s.add(i2==RotateRight (i1, i1 & 0xF))
s.add(i3==i2 ^ C2)
s.add(i4==RotateLeft(i3, i3 & 0xF))
s.add(i5==i4 + C3)
s.add(outp==RotateLeft (i5, URem(i5, 60)))

s.add(outp & 0xFFFFFFFF == inp & 0xFFFFFFFF)

print (s.check())
m=s.model()
print (m)
print (" inp=0x%X" % m[inp].as_long())
print ("outp=0x%X" % m[outp].as_long())
```

It is indeed so:

```
sat
[i1 = 14869545517796235860,
 i3 = 8388171335828825253,
 i5 = 6918262285561543945,
 inp = 1370377541658871093,
 outp = 14543180351754208565,
 i4 = 10167065714588685486,
 i2 = 5541032613289652645]
 inp=0x13048F1D12C00535
outp=0xC9D3C17A12C00535
```

Let's be more sadistic and add another constraint: last 16 bits must be `0x1234`:

```
#!/usr/bin/env python3

from z3 import *

C1=0x5D7E0D1F2E0F1F84
C2=0x388D76AEE8CB1500
C3=0xD2E9EE7E83C4285B

inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i6 outp', 64)

s = Solver()
s.add(i1==inp*C1)
s.add(i2==RotateRight (i1, i1 & 0xF))
s.add(i3==i2 ^ C2)
s.add(i4==RotateLeft(i3, i3 & 0xF))
s.add(i5==i4 + C3)
s.add(outp==RotateLeft (i5, URem(i5, 60)))

s.add(outp & 0xFFFFFFFF == inp & 0xFFFFFFFF)
s.add(outp & 0xFFFF == 0x1234)

print (s.check())
m=s.model()
print (m)
print (" inp=0x%X" % m[inp].as_long())
print ("outp=0x%X" % m[outp].as_long())
```

Oh yes, this possible as well:

```
sat
[i1 = 2834222860503985872,
 i3 = 2294680776671411152,
 i5 = 17492621421353821227,
 inp = 461881484695179828,
 outp = 419247225543463476,
 i4 = 2294680776671411152,
 i2 = 2834222860503985872]
 inp=0x668EEC35F961234
outp=0x5D177215F961234
```

Z3 works very fast and it implies that the algorithm is weak, it is not cryptographic at all (like the most of the amateur cryptography).

19.3 Swapping encryption and decryption procedures

By the way, this is quite important property of symmetric cryptography: encryption and decryption procedures can be swapped without loss of security:

```
# copypasted from https://pypi.org/project/pycrypto/ and reworked:

from Crypto.Cipher import AES

obj = AES.new('This is a key123', AES.MODE_CBC, 'This is an IV456')
message = "Hello, world!!!!" # message's size must be multiple of 16

# decrypt "Hello, world" message:
rubbish = obj.decrypt(message)

print rubbish # rubbish printed

# encrypt it back:

obj2 = AES.new('This is a key123', AES.MODE_CBC, 'This is an IV456')
plaintext_again=obj2.encrypt(rubbish)

print plaintext_again # "Hello, world" printed
```

The author saw this in shareware checking license keys, etc, when a key is encrypted (yes) to get information from it. Perhaps, their key generator decrypted data when putting it to key file. Hopefully, they swapped functions just by mistake, but it worked, so they left it all as is.

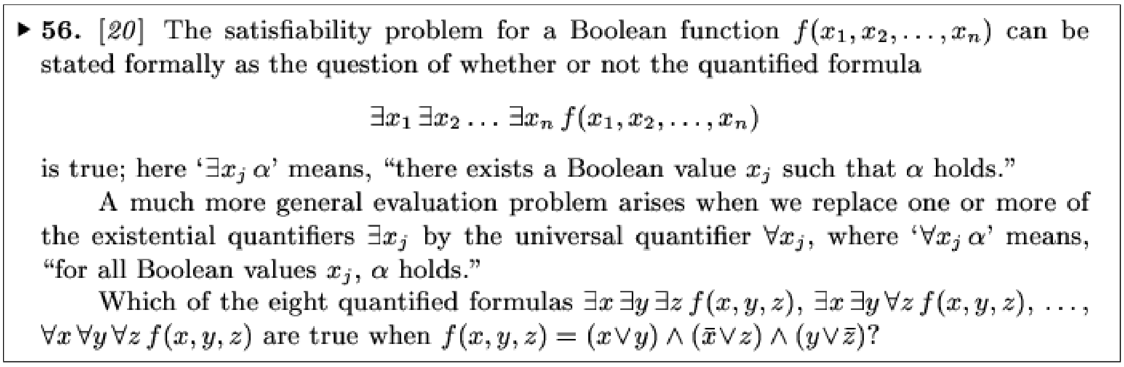
There is also a case of 3DES or Triple DES. Several variants are available, and the most popular encryption procedure (DES-EDE3) is: 1) encrypt with key1; 2) decrypt with key2; 3) encrypt with key3. The decryption procedure is inverted. This procedure is as secure as triple encryption using 3 keys (DES-EEE3).

Chapter 20

First-Order Logic

20.1 Exercise 56 from TAOCP "7.1.1 Boolean Basics", solving it using Z3

Page 41 from fasc0b.ps or http://www.cs.utsa.edu/~wagner/knuth/fasc0b.pdf.



▸ **56.** [*20*] The satisfiability problem for a Boolean function $f(x_1, x_2, \ldots, x_n)$ can be stated formally as the question of whether or not the quantified formula

$$\exists x_1\, \exists x_2 \ldots \exists x_n\, f(x_1, x_2, \ldots, x_n)$$

is true; here '$\exists x_j\, \alpha$' means, "there exists a Boolean value x_j such that α holds."

A much more general evaluation problem arises when we replace one or more of the existential quantifiers $\exists x_j$ by the universal quantifier $\forall x_j$, where '$\forall x_j\, \alpha$' means, "for all Boolean values x_j, α holds."

Which of the eight quantified formulas $\exists x\, \exists y\, \exists z\, f(x, y, z)$, $\exists x\, \exists y\, \forall z\, f(x, y, z)$, ..., $\forall x\, \forall y\, \forall z\, f(x, y, z)$ are true when $f(x, y, z) = (x \vee y) \wedge (\bar{x} \vee z) \wedge (y \vee \bar{z})$?

Figure 20.1: Page 41

For exists/forall/forall:

```
(assert
        (exists ((x Bool)) (forall ((y Bool)) (forall ((z Bool))
                (and
                        (or x y)
                        (or (not x) z)
                        (or y (not z))
                )))
        )
)
(check-sat)
```

All the rest: https://smt.st/current_tree/FOL/TAOCP_7_1_1_exercise_56.
Results:

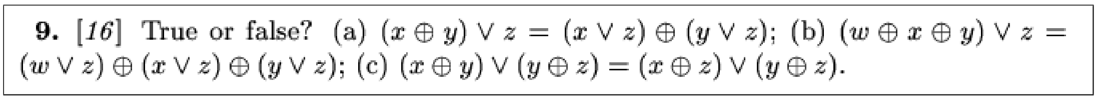
```
z3 -smt2 KnuthAAA.smt
z3 -smt2 KnuthAAE.smt
z3 -smt2 KnuthAEA.smt
z3 -smt2 KnuthAEE.smt
z3 -smt2 KnuthEAA.smt
z3 -smt2 KnuthEAE.smt
z3 -smt2 KnuthEEA.smt
z3 -smt2 KnuthEEE.smt
```

```
unsat
unsat
```

```
unsat
sat
unsat
unsat
sat
sat
```

20.2 Exercise 9 from TAOCP "7.1.1 Boolean Basics", solving it using Z3

Page 34 from fasc0b.ps or http://www.cs.utsa.edu/~wagner/knuth/fasc0b.pdf.



9. [*16*] True or false? (a) $(x \oplus y) \vee z = (x \vee z) \oplus (y \vee z)$; (b) $(w \oplus x \oplus y) \vee z = (w \vee z) \oplus (x \vee z) \oplus (y \vee z)$; (c) $(x \oplus y) \vee (y \oplus z) = (x \oplus z) \vee (y \oplus z)$.

Figure 20.2: Page 34

For (a):

```
(assert
        (forall ((x Bool) (y Bool) (z Bool))
                (=
                        (or (xor x y) z)
                        (xor (or x z) (or y z))
                )
        )
)
(check-sat)
```

For (b):

```
(assert
        (forall ((x Bool) (y Bool) (z Bool) (w Bool))
                (=
                        (or (xor w x y) z)
                        (xor (or w z) (or x z) (or y z))
                )
        )
)
(check-sat)
```

For (c):

```
(assert
        (forall ((x Bool) (y Bool) (z Bool))
                (=
                        (or (xor x y) (xor y z))
                        (or (xor x z) (xor y z))
                )
        )
)
(check-sat)
```

Results:

```
% z3 -smt2 Knuth_a.smt
unsat
% z3 -smt2 Knuth_b.smt
sat
% z3 -smt2 Knuth_c.smt
sat
```

Chapter 21

Cellular automata

21.1 Conway's "Game of Life"

21.1.1 Reversing back the state of "Game of Life"

How could we reverse back a known state of GoL? This can be solved by brute-force, but this is extremely slow and inefficient.

Let's try to use SAT solver.

First, we need to define a function which will tell, if the new cell will be created/born, preserved/stay or died. Quick refresher: cell is born if it has 3 neighbours, it stays alive if it has 2 or 3 neighbours, it dies in any other case.

This is how I can define a function reflecting state of a new cell in the next state:

```
if center==true:
        return popcnt2(neighbours) || popcnt3(neighbours)
if center==false
        return popcnt3(neighbours)
```

We can get rid of "if" construction:

```
result=(center==true && (popcnt2(neighbours) || popcnt3(neighbours))) || (center==
    false && popcnt3(neighbours))
```

...where "center" is state of a center cell, "neighbours" are 8 neighbouring cells, popcnt2 is a function which returns True if it has exactly 2 bits on input, popcnt3 is the same, but for 3 bits (just like these were used in my "Minesweeper" example (3.10.2)).

Using Wolfram Mathematica, I first create all helper functions and truth table for the function, which returns *true*, if a cell must be present in the next state, or *false* if not:

```
In[1]:= popcount[n_Integer]:=IntegerDigits[n,2] // Total

In[2]:= popcount2[n_Integer]:=Equal[popcount[n],2]

In[3]:= popcount3[n_Integer]:=Equal[popcount[n],3]

In[4]:= newcell[center_Integer,neighbours_Integer]:=(center==1 && (popcount2[
    neighbours]|| popcount3[neighbours]))||
(center==0 && popcount3[neighbours])

In[13]:= NewCellIsTrue=Flatten[Table[Join[{center},PadLeft[IntegerDigits[neighbours
    ,2],8]] ->
Boole[newcell[center, neighbours]],{neighbours,0,255},{center,0,1}]]

Out[13]= {{0,0,0,0,0,0,0,0,0}->0,
{1,0,0,0,0,0,0,0,0}->0,
{0,0,0,0,0,0,0,0,1}->0,
{1,0,0,0,0,0,0,0,1}->0,
{0,0,0,0,0,0,0,1,0}->0,
{1,0,0,0,0,0,0,1,0}->0,
{0,0,0,0,0,0,0,1,1}->0,
{1,0,0,0,0,0,0,1,1}->1,

...
```

Now we can create a CNF-expression out of truth table:

```
In[14]:= BooleanConvert[BooleanFunction[NewCellIsTrue,{center,a,b,c,d,e,f,g,h}],"CNF
    "]
Out[14]= (!a||!b||!c||!d)&&(!a||!b||!c||!e)&&(!a||!b||!c||!f)&&(!a||!b||!c||!g)&&(!a
    ||!b||!c||!h)&&
(!a||!b||!d||!e)&&(!a||!b||!d||!f)&&(!a||!b||!d||!g)&&(!a||!b||!d||!h)&&(!a||!b||!e
    ||!f)&&
(!a||!b||!e||!g)&&(!a||!b||!e||!h)&&(!a||!b||!f||!g)&&(!a||!b||!f||!h)&&(!a||!b||!g
    ||!h)&&
(!a||!c||!d||!e)&&(!a||!c||!d||!f)&&(!a||!c||!d||!g)&&(!a||!c||!d||!h)&&(!a||!c||!e
    ||!f)&&
(!a||!c||!e||!g)&&(!a||!c||!e||!h)&&(!a||!c||!f||!g)&&(!a||!c||!f||!h)&&

...
```

Also, we need a second function, *inverted one*, which will return *true* if the cell must be absent in the next state, or *false* otherwise:

```
In[15]:= NewCellIsFalse=Flatten[Table[Join[{center},PadLeft[IntegerDigits[neighbours
    ,2],8]] ->
Boole[Not[newcell[center, neighbours]]],{neighbours,0,255},{center,0,1}]]
Out[15]= {{0,0,0,0,0,0,0,0,0}->1,
{1,0,0,0,0,0,0,0,0}->1,
{0,0,0,0,0,0,0,0,1}->1,
{1,0,0,0,0,0,0,0,1}->1,
{0,0,0,0,0,0,0,1,0}->1,

...

In[16]:= BooleanConvert[BooleanFunction[NewCellIsFalse,{center,a,b,c,d,e,f,g,h}],"CNF
    "]
Out[16]= (!a||!b||!c||d||e||f||g||h)&&(!a||!b||c||!d||e||f||g||h)&&(!a||!b||c||d||!e
    ||f||g||h)&&
(!a||!b||c||d||e||!f||g||h)&&(!a||!b||c||d||e||f||!g||h)&&(!a||!b||c||d||e||f||g||!h)
    &&
(!a||!b||!center||d||e||f||g||h)&&(!a||b||!c||!d||e||f||g||h)&&(!a||b||!c||d||!e||f||
    g||h)&&
(!a||b||!c||d||e||!f||g||h)&&(!a||b||!c||d||e||f||!g||h)&&(!a||b||!c||d||e||f||g||!h)
    &&
(!a||b||c||!d||!e||f||g||h)&&(!a||b||c||!d||e||!f||g||h)&&(!a||b||c||!d||e||f||!g||h)
    &&

...
```

Using the very same way as in my "Minesweeper" example (3.10.2), I can convert CNF expression to list of clauses:

```
def mathematica_to_CNF (s:str, d:Dict[str, str]) -> List[List[str]]:
    for k in d.keys():
        s=s.replace(k, d[k])
    s=s.replace("!", "-").replace("||", " ").replace("(", "").replace(")", "")
    lst=s.split ("&&")
    rt=[]
    for l in lst:
        rt.append(l.split(" "))
    return rt
```

And again, as in "Minesweeper", there is an invisible border, to make processing simpler. SAT variables are also numbered as in previous example:

```
 1    2   3   4   5   6   7   8   9  10  11
12   13  14  15  16  17  18  19  20  21  22
23   24  25  26  27  28  29  30  31  32  33
34   35  36  37  38  39  40  41  42  43  44

...

100 101 102 103 104 105 106 107 108 109 110
111 112 113 114 115 116 117 118 119 120 121
```

Also, there is a visible border, always fixed to *False*, to make things simpler.

Now the working source code. Whenever we encounter "*" in `final_state[]`, we add clauses generated by `cell_is_true()` function, or `cell_is_false()` if otherwise. When we get a solution, it is negated and added to the list of clauses, so when minisat is executed next time, it will skip solution which was already printed.

```
...

def cell_is_false (center, a):
    s="(!a||!b||!c||d||e||f||g||h)&&(!a||!b||c||!d||e||f||g||h)&&(!a||!b||c||d||!e||f
        ||g||h)&&" \
      "(!a||!b||c||d||e||!f||g||h)&&(!a||!b||c||d||e||f||!g||h)&&(!a||!b||c||d||e||f
         ||g||!h)&&" \
      "(!a||!b||!center||d||e||f||g||h)&&(!a||b||!c||!d||e||f||g||h)&&(!a||b||!c||d
         ||!e||f||g||h)&&" \
      "(!a||b||!c||d||e||!f||g||h)&&(!a||b||!c||d||e||f||!g||h)&&(!a||b||!c||d||e||f
         ||g||!h)&&" \
      "(!a||b||c||!d||!e||f||g||h)&&(!a||b||c||!d||e||!f||g||h)&&(!a||b||c||!d||e||f
         ||!g||h)&&" \
      "(!a||b||c||!d||e||f||g||!h)&&(!a||b||c||d||!e||!f||g||h)&&(!a||b||c||d||!e||f
         ||!g||h)&&" \
      "(!a||b||c||d||!e||f||g||!h)&&(!a||b||c||d||e||!f||!g||h)&&(!a||b||c||d||e||!f
         ||g||!h)&&" \
      "(!a||b||c||d||e||f||!g||!h)&&(!a||!c||!center||d||e||f||g||h)&&(!a||c||!center
         ||!d||e||f||g||h)&&" \
      "(!a||c||!center||d||!e||f||g||h)&&(!a||c||!center||d||e||!f||g||h)&&(!a||c||!
         center||d||e||f||!g||h)&&" \
      "(!a||c||!center||d||e||f||g||!h)&&(a||!b||!c||!d||e||f||g||h)&&(a||!b||!c||d
         ||!e||f||g||h)&&" \
      "(a||!b||!c||d||e||!f||g||h)&&(a||!b||!c||d||e||f||!g||h)&&(a||!b||!c||d||e||f
         ||g||!h)&&" \
      "(a||!b||c||!d||!e||f||g||h)&&(a||!b||c||!d||e||!f||g||h)&&(a||!b||c||!d||e||f
         ||!g||h)&&" \
      "(a||!b||c||!d||e||f||g||!h)&&(a||!b||c||d||!e||!f||g||h)&&(a||!b||c||d||!e||f
         ||!g||h)&&" \
      "(a||!b||c||d||!e||f||g||!h)&&(a||!b||c||d||e||!f||!g||h)&&(a||!b||c||d||e||!f
         ||g||!h)&&" \
      "(a||!b||c||d||e||f||!g||!h)&&(a||b||!c||!d||!e||f||g||h)&&(a||b||!c||!d||e||!f
         ||g||h)&&" \
      "(a||b||!c||!d||e||f||!g||h)&&(a||b||!c||!d||e||f||g||!h)&&(a||b||!c||d||!e||!f
         ||g||h)&&" \
      "(a||b||!c||d||!e||f||!g||h)&&(a||b||!c||d||!e||f||g||!h)&&(a||b||!c||d||e||!f
         ||!g||h)&&" \
      "(a||b||!c||d||e||!f||g||!h)&&(a||b||!c||d||e||f||!g||!h)&&(a||b||c||!d||!e||!f
         ||g||h)&&" \
      "(a||b||c||!d||!e||f||!g||h)&&(a||b||c||!d||!e||f||g||!h)&&(a||b||c||!d||e||!f
         ||!g||h)&&" \
      "(a||b||c||!d||e||!f||g||!h)&&(a||b||c||!d||e||f||!g||!h)&&(a||b||c||d||!e||!f
         ||!g||h)&&" \
      "(a||b||c||d||!e||!f||g||!h)&&(a||b||c||d||!e||f||!g||!h)&&(a||b||c||d||e||!f
         ||!g||!h)&&" \
      "(!b||!c||!center||d||e||f||g||h)&&(!b||c||!center||!d||e||f||g||h)&&(!b||c||!
         center||d||!e||f||g||h)&&" \
      "(!b||c||!center||d||e||!f||g||h)&&(!b||c||!center||d||e||f||!g||h)&&(!b||c||!
         center||d||e||f||g||!h)&&" \
      "(b||!c||!center||!d||e||f||g||h)&&(b||!c||!center||d||!e||f||g||h)&&(b||!c||!
         center||d||e||!f||g||h)&&" \
      "(b||!c||!center||d||e||f||!g||h)&&(b||!c||!center||d||e||f||g||!h)&&(b||c||!
         center||!d||!e||f||g||h)&&" \
      "(b||c||!center||!d||e||!f||g||h)&&(b||c||!center||!d||e||f||!g||h)&&(b||c||!
         center||!d||e||f||g||!h)&&" \
      "(b||c||!center||d||!e||!f||g||h)&&(b||c||!center||d||!e||f||!g||h)&&(b||c||!
         center||d||!e||f||g||!h)&&" \
      "(b||c||!center||d||e||!f||!g||h)&&(b||c||!center||d||e||!f||g||!h)&&(b||c||!
         center||d||e||f||!g||!h)"

    return mathematica_to_CNF(s, center, a)

def cell_is_true (center, a):
    s="(!a||!b||!c||!d)&&(!a||!b||!c||!e)&&(!a||!b||!c||!f)&&(!a||!b||!c||!g)&&(!a||!
        b||!c||!h)&&" \
      "(!a||!b||!d||!e)&&(!a||!b||!d||!f)&&(!a||!b||!d||!g)&&(!a||!b||!d||!h)&&(!a||!
         b||!e||!f)&&" \
      "(!a||!b||!e||!g)&&(!a||!b||!e||!h)&&(!a||!b||!f||!g)&&(!a||!b||!f||!h)&&(!a||!
         b||!g||!h)&&" \
      "(!a||!c||!d||!e)&&(!a||!c||!d||!f)&&(!a||!c||!d||!g)&&(!a||!c||!d||!h)&&(!a||!
         c||!e||!f)&&" \
      "(!a||!c||!e||!g)&&(!a||!c||!e||!h)&&(!a||!c||!f||!g)&&(!a||!c||!f||!h)&&(!a||!
         c||!g||!h)&&" \
      "(!a||!d||!e||!f)&&(!a||!d||!e||!g)&&(!a||!d||!e||!h)&&(!a||!d||!f||!g)&&(!a||!
         d||!f||!h)&&" \
      "(!a||!d||!g||!h)&&(!a||!e||!f||!g)&&(!a||!e||!f||!h)&&(!a||!e||!g||!h)&&(!a||!
         f||!g||!h)&&" \
      "(a||b||c||center||d||e||f)&&(a||b||c||center||d||e||g)&&(a||b||c||center||d||e
         ||h)&&" \
      "(a||b||c||center||d||f||g)&&(a||b||c||center||d||f||h)&&(a||b||c||center||d||g
         ||h)&&" \
      "(a||b||c||center||e||f||g)&&(a||b||c||center||e||f||h)&&(a||b||c||center||e||g
         ||h)&&" \
      "(a||b||c||center||f||g||h)&&(a||b||c||d||e||f||g)&&(a||b||c||d||e||f||h)&&(a||
         b||c||d||e||g||h)&&" \
      "(a||b||c||d||f||g||h)&&(a||b||c||e||f||g||h)&&(a||b||center||d||e||f||g)&&(a||
         b||center||d||e||f||h)&&" \
      "(a||b||center||d||e||g||h)&&(a||b||center||d||f||g||h)&&(a||b||center||e||f||g
         ||h)&&" \
      "(a||b||d||e||f||g||h)&&(a||c||center||d||e||f||g)&&(a||c||center||d||e||f||h)
         &&" \
      "(a||c||center||d||e||g||h)&&(a||c||center||d||f||g||h)&&(a||c||center||e||f||g
         ||h)&&" \
      "(a||c||d||e||f||g||h)&&(a||center||d||e||f||g||h)&&(!b||!c||!d||!e)&&(!b||!c
         ||!d||!f)&&" \
      "(!b||!c||!d||!g)&&(!b||!c||!d||!h)&&(!b||!c||!e||!f)&&(!b||!c||!e||!g)&&(!b||!
         c||!e||!h)&&" \
      "(!b||!c||!f||!g)&&(!b||!c||!f||!h)&&(!b||!c||!g||!h)&&(!b||!d||!e||!f)&&(!b||!
         d||!e||!g)&&" \
      "(!b||!d||!e||!h)&&(!b||!d||!f||!g)&&(!b||!d||!f||!h)&&(!b||!d||!g||!h)&&(!b||!
         e||!f||!g)&&" \
      "(!b||!e||!f||!h)&&(!b||!e||!g||!h)&&(!b||!f||!g||!h)&&(b||c||center||d||e||f||
         g)&&" \
      "(b||c||center||d||e||f||h)&&(b||c||center||d||e||g||h)&&(b||c||center||d||f||g
         ||h)&&" \
      "(b||c||center||e||f||g||h)&&(b||c||d||e||f||g||h)&&(b||center||d||e||f||g||h)
         &&" \
      "(!c||!d||!e||!f)&&(!c||!d||!e||!g)&&(!c||!d||!e||!h)&&(!c||!d||!f||!g)&&(!c||!
         d||!f||!h)&&" \
      "(!c||!d||!g||!h)&&(!c||!e||!f||!g)&&(!c||!e||!f||!h)&&(!c||!e||!g||!h)&&(!c||!
         f||!g||!h)&&" \
      "(c||center||d||e||f||g||h)&&(!d||!e||!f||!g)&&(!d||!e||!f||!h)&&(!d||!e||!g||!
         h)&&(!d||!f||!g||!h)&&" \
      "(!e||!f||!g||!h)"

    return mathematica_to_CNF(s, center, a)

...
```

(https://smt.st/current_tree/CA/GoL/GoL_SAT_utils.py)

```
#!/usr/bin/python3

import os
from GoL_SAT_utils import *

final_state=[
" * ",
"* *",
" * "]

H=len(final_state) # HEIGHT
W=len(final_state[0]) # WIDTH

print ("HEIGHT=", H, "WIDTH=", W)

s=SAT_lib.SAT_lib(maxsat=False)
VAR_FALSE=s.const_false
grid=[[s.create_var() for w in range(W)] for h in range(H)]

def try_again ():
    # rules for the main part of grid
    for r in range(H):
        for c in range(W):
            if final_state[r][c]=="*":
                s.add_clauses (cell_is_true(coords_to_var(grid, VAR_FALSE, r, c, H, W
                    ),
                                             get_neighbours(grid, VAR_FALSE, r, c, H,
                                                 W)))
            else:
                s.add_clauses (cell_is_false(coords_to_var(grid, VAR_FALSE, r, c, H,
                    W),
                                              get_neighbours(grid, VAR_FALSE, r, c, H
                                                 , W)))

    # cells behind visible grid must always be false:
    for c in range(-1, W+1):
        for r in [-1,H]:
            s.add_clauses (cell_is_false(coords_to_var(grid, VAR_FALSE, r, c, H, W),
                                          get_neighbours(grid, VAR_FALSE, r, c, H, W)
                                              ))
    for c in [-1,W]:
        for r in range(-1, H+1):
            s.add_clauses (cell_is_false(coords_to_var(grid, VAR_FALSE, r, c, H, W),
                                          get_neighbours(grid, VAR_FALSE, r, c, H, W)
                                              ))

    assert s.solve()
    tmp=SAT_solution_to_grid(grid, VAR_FALSE, s.solution, H, W)

    print_grid(tmp)
    write_RLE(tmp)

    return tmp

while True:
    try_again()
    if s.fetch_next_solution()==False:
        break
    print ("")
```

(https://smt.st/current_tree/CA/GoL/reverse1.py)

Here is the result:

```
HEIGHT= 3 WIDTH= 3
.*.
*.*
.*.
1.rle written

.**
*..
*.*
2.rle written

**.
..*
*.*
3.rle written

*.*
*..
.**
4.rle written

*.*
..*
**.
5.rle written

*.*
.*.
*.*
6.rle written

unsat!
```

The first result is the same as initial state. Indeed: this is “still life”, i.e., state which will never change, and it is correct solution. The last solution is also valid.

Now the problem: 2nd, 3rd, 4th and 5th solutions are equivalent to each other, they just mirrored or rotated. In fact, this is reflectional[1] (like in mirror) and rotational[2] symmetries. We can solve this easily: we will take each solution, reflect and rotate it and add them negated to the list of clauses, so minisat will skip them during its work:

```

...

while True:
    solution=try_again()
```

[1] https://en.wikipedia.org/wiki/Reflection_symmetry

[2] https://en.wikipedia.org/wiki/Rotational_symmetry

```
    if solution==None:
        break

    s.add_clause(SAT_lib.negate_clause(GoL_SAT_utils.grid_to_clause(solution, grid,
        VAR_FALSE, H, W)))
    s.add_clause(SAT_lib.negate_clause(GoL_SAT_utils.grid_to_clause(my_utils.
        reflect_vertically(solution), grid, VAR_FALSE, H, W)))
    s.add_clause(SAT_lib.negate_clause(GoL_SAT_utils.grid_to_clause(my_utils.
        reflect_horizontally(solution), grid, VAR_FALSE, H, W)))
    # is this square?
    if W==H:
        s.add_clause(SAT_lib.negate_clause(GoL_SAT_utils.grid_to_clause(my_utils.
            rotate_rect_array(solution,1), grid, VAR_FALSE, H, W)))
        s.add_clause(SAT_lib.negate_clause(GoL_SAT_utils.grid_to_clause(my_utils.
            rotate_rect_array(solution,2), grid, VAR_FALSE, H, W)))
        s.add_clause(SAT_lib.negate_clause(GoL_SAT_utils.grid_to_clause(my_utils.
            rotate_rect_array(solution,3), grid, VAR_FALSE, H, W)))

...
```

(https://smt.st/current_tree/CA/GoL/reverse2.py)

Functions `reflect_vertically()`, `reflect_horizontally` and `rotate_squarearray()` are simple array manipulation routines.

Now we get just 3 solutions:

```
HEIGHT= 3 WIDTH= 3
.*.
*.*
.*.
1.rle written

.**
*..
*.*
2.rle written

*.*
.*.
*.*
3.rle written

unsat!
```

This one has only one single ancestor:

```
final_state=[
" * ",
" * ",
" * "]
_PRE_END

_PRE_BEGIN
HEIGHT= 3 WIDTH= 3
...
***
...
1.rle written

unsat!
```

This is oscillator, of course.

How many states can lead to such picture?

```
final_state=[
"  *  ",
"     ",
" **  ",
"  *  ",
"  *  ",
" *** "]
```

28, these are few:

```
HEIGHT= 6 WIDTH= 5
.*.*.
..*..
.**.*
..*..
..*.*
.**..
1.rle written

.*.*.
..*..
.**.*
..*..
*.*.*
.**..
2.rle written

..*.*
..**.
.**..
....*
*.*.*
.**..
3.rle written

..*.*
..**.
.**..
*...*
..*.*
.**..
4.rle written

...
```

Now the biggest, “space invader”:

```
final_state=[
"             ",
"   *     *   ",
"    *   *    ",
"   *******   ",
"  ** *** **  ",
" *********** ",
" * ******* * ",
" * *     * * ",
"    ** **    ",
"             "]
```

```
HEIGHT= 10 WIDTH= 13
..*.*.**.....
.....*****...
```

```
....**..*....
......*...*..
..**...*.*...
.*..*.*.**..*
*....*....*.*
..*.*..*.....
..*.....*.*..
....**..*.*..
1.rle written

*.*.*.**.....
.....*****...
....**..*....
......*...*..
..**...*.*...
.*..*.*.**..*
*....*....*.*
..*.*..*.....
..*.....*.*..
....**..*.*..
2.rle written

..*.*.**.....
*....*****...
....**..*....
......*...*..
..**...*.*...
.*..*.*.**..*
*....*....*.*
..*.*..*.....
..*.....*.*..
....**..*.*..
3.rle written

...
```

I don't know how many possible states can lead to "space invader", perhaps, too many. Had to stop it. And it slows down during execution, because number of clauses is increasing (because of negating solutions addition).

All solutions are also exported to RLE files, which can be opened by Golly[3].

Garden of Eden

As they say, "A Garden of Eden is a pattern that has no parents and thus can only occur in generation 0."[4].

We can check if these are really *gardens*:

```
#!/usr/bin/python3

from typing import List
import os, SAT_lib, GoL_SAT_utils, my_utils

# https://www.conwaylife.com/wiki/Garden_of_Eden#Records
# https://conwaylife.com/patterns/gardenofeden1.cells
# 1971, Roger Banks et al., 33 × 9 = 297
"""
final_state=[
"OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO",
"OO.O.OOO.OOO.OO.O.O.O.O.O.O.O.O.O",
"O.O.OOO.OOO.OOOO.OOO.O.O.O.O.O.O.",
"OOOOO.OOO.OOO.OOOO.OOOOOOOOOOOOOO",
"O.O.OO.OOO.OOO.O.OOO.O.O.O.O.O.O.",
```

[3] http://golly.sourceforge.net/

[4] https://www.conwaylife.com/wiki/Garden_of_Eden

```
"OOOO.OOO.OOO.OOOOO.OO.O.O.O.O.O.O",
".OO.OOO.OOO.OOO.O.O.OOOOOOOOOOOOO",
"OO.OO.OOO.OOO.OO.OOOO.O.O.O.O.O.O",
"OOOOOOOOOOOOOOOOOO.OOOOOOOOOOOOOO"]
"""

# https://www.conwaylife.com/wiki/Garden_of_Eden#Records
# https://conwaylife.com/patterns/gardenofeden5x45.cells
# 2016, Steven Eker, 45 × 5 = 225
"""
final_state=[
"OO.O.O..O.OOO.O..OOO.OOOOOOO..OOOOOO..O.O.O.O",
".OO.O.OO.O.O.O.O.O.OOO.O.O.O.O.O.O.OOO.O.O.O.",
"..OO.OO..OOOO..OOOOOO..OOOOOO..OOOOO.OOOOO.O.",
".OO.O.OO.O.O.O.O.O.OOO.O.O.O.O.O.O.OOO.O.O.O.",
"OO.O.O..O.OOO.O..OOO.OOOOOOO..OOOOOO..O.O.O.O"]
"""
# https://www.conwaylife.com/wiki/Garden_of_Eden#Records
# https://www.conwaylife.com/patterns/gardenofeden11.cells
# 2017, Steven Eker, 9 × 11 = 99
final_state=[
"..OO.OOO.O.",
".O..O.O.O..",
"O.OOO..OOO.",
".O.O.O..O.O",
".OOO..OO..O",
"OO...OO..O.",
"O.O.O.OOO..",
"O..O.O..O..",
".OOO..OOOO."]

H=len(final_state) # HEIGHT
W=len(final_state[0]) # WIDTH

print ("HEIGHT=", H, "WIDTH=", W)

s=SAT_lib.SAT_lib(maxsat=False)
VAR_FALSE=s.const_false
grid=[[s.create_var() for w in range(W)] for h in range(H)]

def try_again ():
    # rules for the main part of grid
    for r in range(H):
        for c in range(W):
            if final_state[r][c]=="O":
                s.add_clauses (GoL_SAT_utils.cell_is_true(GoL_SAT_utils.coords_to_var
                    (grid, VAR_FALSE, r, c, H, W),
                                                 GoL_SAT_utils.get_neighbours(grid,
                                                     VAR_FALSE, r, c, H, W)))
            else:
                s.add_clauses (GoL_SAT_utils.cell_is_false(GoL_SAT_utils.
                    coords_to_var(grid, VAR_FALSE, r, c, H, W),
                                                  GoL_SAT_utils.get_neighbours(grid,
                                                      VAR_FALSE, r, c, H, W)))

    # cells behind visible grid must always be false:
    for c in range(-1, W+1):
        for r in [-1,H]:
            s.add_clauses (GoL_SAT_utils.cell_is_false(GoL_SAT_utils.coords_to_var(
                grid, VAR_FALSE, r, c, H, W),
                                          GoL_SAT_utils.get_neighbours(grid,
                                             VAR_FALSE, r, c, H, W)))
    for c in [-1,W]:
        for r in range(-1, H+1):
            s.add_clauses (GoL_SAT_utils.cell_is_false(GoL_SAT_utils.coords_to_var(
               grid, VAR_FALSE, r, c, H, W),
                                          GoL_SAT_utils.get_neighbours(grid,
                                             VAR_FALSE, r, c, H, W)))

    if s.solve()==False:
        return None

    tmp=GoL_SAT_utils.SAT_solution_to_grid(grid, VAR_FALSE, s.solution, H, W)

    GoL_SAT_utils.print_grid(tmp)
    GoL_SAT_utils.write_RLE(tmp)

    return tmp

solution=try_again()
if solution==None:
    print ("Eden (UNSAT)")
else:
    print ("not Eden (SAT)")
```

It's indeed so for the *garden* I've copypasted.

21.1.2 Finding "still lives"

"Still life" in terms of GoL is a state which doesn't change at all.

... using SAT-solver

First, using previous definitions, we will define a truth table of function, which will return *true*, if the center cell of the next state is the same as it has been in the previous state, i.e., hasn't been changed:

```
In[17]:= stillife=Flatten[Table[Join[{center},PadLeft[IntegerDigits[neighbours
   ,2],8]]->
Boole[Boole[newcell[center,neighbours]]==center],{neighbours,0,255},{center,0,1}]]
Out[17]= {{0,0,0,0,0,0,0,0,0}->1,
{1,0,0,0,0,0,0,0,0}->0,
{0,0,0,0,0,0,0,0,1}->1,
{1,0,0,0,0,0,0,0,1}->0,

...

In[18]:= BooleanConvert[BooleanFunction[stillife,{center,a,b,c,d,e,f,g,h}],"CNF"]
Out[18]= (!a||!b||!c||!center||!d)&&(!a||!b||!c||!center||!e)&&(!a||!b||!c||!center
   ||!f)&&
(!a||!b||!c||!center||!g)&&(!a||!b||!c||!center||!h)&&(!a||!b||!c||center||d||e||f||g
   ||h)&&
(!a||!b||c||center||!d||e||f||g||h)&&(!a||!b||c||center||d||!e||f||g||h)&&(!a||!b||c
   ||center||d||e||!f||g||h)&&
(!a||!b||c||center||d||e||f||!g||h)&&(!a||!b||c||center||d||e||f||g||!h)&&(!a||!b||!
   center||!d||!e)&&

...
```

```
#!/usr/bin/python3

import os, SAT_lib, GoL_SAT_utils, SL_common, my_utils

W=3 # WIDTH
```

```
H=3 # HEIGHT

s=SAT_lib.SAT_lib()
VAR_FALSE=s.const_false
grid=[[s.create_var() for w in range(W)] for h in range(H)]

def try_again ():
    # rules for the main part of grid
    for r in range(H):
        for c in range(W):
            s.add_clauses (SL_common.gen_SL(GoL_SAT_utils.coords_to_var(grid,
               VAR_FALSE, r, c, H, W),
                                               GoL_SAT_utils.get_neighbours(grid,
                                                  VAR_FALSE, r, c, H, W)))

    # cells behind visible grid must always be false:
    for c in range(-1, W+1):
        for r in [-1,H]:
            s.add_clauses (GoL_SAT_utils.cell_is_false(GoL_SAT_utils.coords_to_var(
               grid, VAR_FALSE, r, c, H, W),
                                                      GoL_SAT_utils.get_neighbours(
                                                         grid, VAR_FALSE, r, c, H, W)
                                                         ))
    for c in [-1,W]:
        for r in range(-1, H+1):
            s.add_clauses (GoL_SAT_utils.cell_is_false(GoL_SAT_utils.coords_to_var(
               grid, VAR_FALSE, r, c, H, W),
                                                      GoL_SAT_utils.get_neighbours(
                                                         grid, VAR_FALSE, r, c, H, W)
                                                         ))
    if s.solve()==False:
        return None

    t=GoL_SAT_utils.SAT_solution_to_grid(grid, VAR_FALSE, s.solution, H, W)
    GoL_SAT_utils.print_grid(t)
    GoL_SAT_utils.write_RLE(t)

    return t

while True:
    solution=try_again()
    if solution==None:
        break

    s.add_clause(SAT_lib.negate_clause(GoL_SAT_utils.grid_to_clause(solution, grid,
       VAR_FALSE, H, W)))
    s.add_clause(SAT_lib.negate_clause(GoL_SAT_utils.grid_to_clause(my_utils.
       reflect_vertically(solution), grid, VAR_FALSE, H, W)))
    s.add_clause(SAT_lib.negate_clause(GoL_SAT_utils.grid_to_clause(my_utils.
       reflect_horizontally(solution), grid, VAR_FALSE, H, W)))
    # is this square?
    if W==H:
        s.add_clause(SAT_lib.negate_clause(GoL_SAT_utils.grid_to_clause(my_utils.
           rotate_rect_array(solution,1), grid, VAR_FALSE, H, W)))
        s.add_clause(SAT_lib.negate_clause(GoL_SAT_utils.grid_to_clause(my_utils.
           rotate_rect_array(solution,2), grid, VAR_FALSE, H, W)))
        s.add_clause(SAT_lib.negate_clause(GoL_SAT_utils.grid_to_clause(my_utils.
           rotate_rect_array(solution,3), grid, VAR_FALSE, H, W)))
    print ("")
```

(https://smt.st/current_tree/CA/GoL/SL1.py)

What we've got for $2 \cdot 2$?

```
..
..
1.rle written

**
**
2.rle written

unsat!
```

Both solutions are correct: empty square will progress into empty square (no cells are born). $2 \cdot 2$ box is also known "still life".

What about $3 \cdot 3$ square?

```
...
...
...
1.rle written

.**
.**
...
2.rle written

.**
*.*
**.
3.rle written

.*.
*.*
**.
4.rle written

.*.
*.*
.*.
5.rle written

unsat!
```

Here is a problem: we see familiar $2 \cdot 2$ box, but shifted. This is indeed correct solution, but we don't interested in it, because it has been already seen.

What we can do is add another condition. We can force minisat to find solutions with no empty rows and columns. This is easy. These are SAT variables for $5 \cdot 5$ square:

```
1    2   3   4   5
6    7   8   9  10
11  12  13  14  15
16  17  18  19  20
21  22  23  24  25
```

Each clause is "OR" clause, so all we have to do is to add 5 clauses:

```
1 OR 2 OR 3 OR 4 OR 5
6 OR 7 OR 8 OR 9 OR 10

...
```

That means that each row must have at least one *True* value somewhere. We can also do this for each column as well.

```
...

    # each row must contain at least one cell!
    for r in range(H):
        clauses.append(" ".join([coords_to_var(r, c, H, W) for c in range(W)]))

    # each column must contain at least one cell!
    for c in range(W):
        clauses.append(" ".join([coords_to_var(r, c, H, W) for r in range(H)]))

...
```

(https://smt.st/current_tree/CA/GoL/SL2.py)

Now we can see that $3 \cdot 3$ square has 3 possible "still lives":

```
.*.
*.*
**.
1.rle written

.*.
*.*
.*.
2.rle written

.**
*.*
**.
3.rle written

unsat!
```

$4 \cdot 4$ has 7:

```
..**
...*
***.
*...
1.rle written

..**
..*.
*.*.
**..
2.rle written

..**
.*.*
*.*.
**..
3.rle written

..*.
.*.*
*.*.
**..
4.rle written

.**.
*..*
*.*.
```

```
.*..
5.rle written

..*.
.*.*
*.*.
.*..
6.rle written

.**.
*..*
*..*
.**.
7.rle written

unsat!
```

When I try large squares, like $20 \cdot 20$, funny things happen. First of all, minisat finds solutions not very pleasing aesthetically, but still correct, like:

```
....**.**.**.**.**.*
**..*.**.**.**.**.**
*...................
.*..................
**..................
*...................
.*..................
**..................
*...................
.*..................
**..................
*...................
.*..................
**..................
*...................
.*..................
..*.................
...*................
***.................
*...................
1.rle written

...
```

Indeed: all rows and columns has at least one *True* value.

Then minisat begins to add smaller "still lives" into the whole picture:

```
.**....**...**...**.
.**...*..*.*.*...*..
.......**...*......*
..................**
...**............*..
...*.*...........*..
....*.*........**...
**...*.*...**..*....
*.*...*....*....*...
.*..........****.*..
................*...
..*...**..******....
.*.*..*..*..........
*..*...*.*..****....
***.***..*.*....*...
....*..***.**..**...
```

```
**.*..*.............
.*.**.**..........**
*..*..*..*......*..*
**..**..**......**..
43.rle written
```

In other words, result is a square consisting of smaller “still lives”. It then altering these parts slightly, shifting back and forth. Is it cheating? Anyway, it does it in a strict accordance to rules we defined.

However, when I switch to Parallel Lingeling, things are slightly different:

```
...............**.**
...............*...*
....**..**.**...**..
.....*...*.*..*.*...
.....*.*.*.*.**.*...
...**.*.*...*..*....
..*.*.*......**.....
..**...*............
........*...........
.........*..........
..........*.........
...........*........
............*...**..
.............*..*...
..............*.*...
...............*....
.**.................
..*.................
*...................
**..................
2.rle written
```

It’s still correct.

Anyway, we may want a *denser* picture. We can add a rule: in all 5-cell chunks there must be at least one *True* cell. To achieve this, we just split the whole square by 5-cell chunks and add clause for each:

```

...

    # make result denser:
    lst=[]
    for r in range(H):
        for c in range(W):
            lst.append(GoL_SAT_utils.coords_to_var(grid, VAR_FALSE, r, c, H, W))
    # divide them all by chunks and add to clauses:
    CHUNK_LEN=3
    for c in my_utils.partition(lst,int(len(lst)/CHUNK_LEN)):
        s.add_clause(c)

...
```

(https://smt.st/current_tree/CA/GoL/SL3.py)

This is indeed denser:

```
..**.**......*.*.*..
...*.*.....***.**.*.
...*..*...*.......*.
....*.*..*.*......**
...**.*.*..*...**.*.
..*...*.***.....*.*.
...*.*.*......*..*..
****.*..*....*.**...
*....**.*....*.*....
```

```
...**..*...**..*....
..*..*....*....*.**.
.*.*.**....****.*..*
..*.*....*.*..*..**.
....*.****..*..*.*..
....**....*.*.**..*.
*.**...****.*..*.**.
**...**.....**.*....
...**..*..**..*.**.*
***.*.*..*.*..*.*.**
*....*....*....*....
1.rle written

..**.**......*.*.*..
...*.*.....***.**.*.
...*..*...*.......*.
....*.*..*.*......**
...**.*.*..*...**.*.
..*...*.***.....*.*.
...*.*.*......*..*..
****.*..*....*.**...
*....**.*....*.*....
...**..*...**..*....
..*..*....*....*.**.
.*.*.**....****.*..*
..*.*....*.*..*..**.
....*.****..*..*.*..
....**....*.*.**..*.
*.**...****.*..*.**.
**...**.....**.*....
...**..*.***..*.**.*
***.*..*.*..*.*.*.**
*.......*..**.**....
2.rle written

...
```

Let's try more dense, one mandatory *true* cell per each 4-cell chunk:

```
.**.**...*....**..**
*.*.*...*.*..*..*..*
*....*...*.*..*.**..
.***.*.....*.**.*...
..*.*.....**...*..*.
*......**..*...*.**.
**.....*...*.**.*...
...**...*...**..*...
**.*..*.*......*...*
.*...**.**..***.****
.*....*.*..*..*.*...
**.***...*.**...*.**
.*.*..****.....*..*.
*....*.....**..**.*.
*.***.*..**.*.....**
.*...*..*......**...
...*.*.**......*.***
..**.*.....**......*
*..*.*.**..*.*..***.
**....*.*...*...*...
1.rle written

.**.**...*....**..**
```

```
*.*.*...*.*..*..*..*
*....*...*.*..*.**..
.***.*.....*.**.*...
..*.*.....**...*..*.
*......**..*...*.**.
**.....*...*.**.*...
...**...*...**..*...
**.*..*.*......*...*
.*...**.**..***.****
.*....*.*..*..*.*...
**.***...*.**...*.**
.*.*..****.....*..*.
*....*.....**..**.*.
*.***.*..**.*.....**
.*...*..*......**..*
...*.*.**......*.**.
..**.*.....**....*..
*..*.*.**..*.*...*.*
**....*.*...*.....**
2.rle written

...
```

...and even more: one cell per each 3-cell chunk:

```
**.*..**...**.**....
*.**..*.*...*.*.*.**
....**..*...*...*.*.
.**..*.*.**.*.*.*.*.
..**.*.*...*.**.*.**
*...*.*.**.*....*.*.
**.*..*...*.*.***..*
.*.*.*.***..**...**.
.*.*.*.*..**...*.*..
**.**.*..*...**.*..*
..*...*.**.**.*.*.**
..*.**.*..*.*.*.*...
**.*.*...*..*.*.*...
.*.*...*.**..*..***.
.*..****.*....**...*
..*.*...*..*...*..*.
.**...*.*.**...*.*..
..*..**.*.*...**.**.
..*.*..*..*..*..*..*
.**.**....**..**..**
1.rle written

**.*..**...**.**....
*.**..*.*...*.*.*.**
....**..*...*...*.*.
.**..*.*.**.*.*.*.*.
..**.*.*...*.**.*.**
*...*.*.**.*....*.*.
**.*..*...*.*.***..*
.*.*.*.***..**...**.
.*.*.*.*..**...*.*..
**.**.*..*...**.*..*
..*...*.**.**.*.*.**
..*.**.*..*.*.*.*...
**.*.*...*..*.*.*...
.*.*...*.**..*..***.
.*..****.*....**...*
```

```
..*.*...*..*...*..*.
.**..**.*.**...*.*..
*..*.*..*.*...**.**.
*..*.*.*..*..*..*..*
.**...*...**..**..**
2.rle written

...
```

This is most dense. Unfortunately, it's impossible to construct "still life" with one mandatory *true* cell per each 2-cell chunk.

... using MaxSAT-solver: getting maximum density still life

Using Open-WBO MaxSAT solver, it can find a $13 \cdot 13$ still life for 11 minutes on my ancient Intel Xeon E3-1220 @ 3.10GHz.

```
cells: 90
density: 0.532544
**.**.*.**.**
**.*.***.*.**
...*.....*...
****.**.*.***
*...*.*.*...*
.*.*..*.**.*.
**.**.*..*.**
.*.*..**.*.*.
*..*.*...*..*
****.****.***
.........*...
**.**.**.*.**
**.**.**.**.*
```

(Open-WBO has been switched to the fastest algorithm for the task, "LinearSU" (1st one).

I just maximize all cells, adding them as soft clauses:

```
...

for r in range(H):
    for c in range(W):
        s.add_soft_clause([GoL_SAT_utils.coords_to_var(grid, VAR_FALSE, r, c, H, W)],
            1)

...
```

(https://smt.st/current_tree/CA/GoL/SL_MaxSAT.py)

See also:

- CSPLIB Problem 032: Maximum density still life
- Maximum Density Still Life
- Also, an interesting paper: Chu, G., & Stuckey, P. J. (2012). A complete solution to the Maximum Density Still Life Problem. Artificial Intelligence, 184-185, 1–16. doi:10.1016/j.artint.2012.02.001

Further work: count solutions, eliminating symmetrical. Search for symmetrical still lives, like: https://www.conwaylife.com/wiki/Clips, https://www.conwaylife.com/wiki/Inflected_clips.

21.1.3 The source code

Source code and Wolfram Mathematica notebook: https://smt.st/current_tree/CA/GoL.

21.1.4 Further reading

Marijn Heule is known to be working on applying SAT solvers to GoL: http://conwaylife.com/wiki/Marijn_Heule.

See also: http://conwaylife.com/wiki/SAT_solver, http://conwaylife.com/wiki/Logic_Life_Search.

21.2 One-dimensional cellular automata and Z3 SMT-solver

Remember John Conway's Game of Life? It's a two-dimensional CA. This one is one-dimensional.

It was fully described in Stephen Wolfram's book "New Kind of Science". Including, the famous Rule 110: https://en.wikipedia.org/wiki/Rule_110.

Can we find *oscillators* (repeating states) and *gliders* (repeating and shifted states)?

N.B.: state is *wrapped*: the leftmost invisible cell is a rightmost one and vice versa.

The source code:

```
#!/usr/bin/env python3

from z3 import *

WIDTH=15

def _try (RULE, STATES, WRAPPED, SHIFTED):

    rules=[]
    for i in range(8):
        if ((RULE>>i)&1)==1:
            rules.append(True)
        else:
            rules.append(False)

    rules=rules[::-1]
    #print "rules=", rules

    def f(a, b, c):
        return If(And(a==True, b==True, c==True), rules[7],
            If(And(a==True, b==True, c==False), rules[6],
            If(And(a==True, b==False, c==True), rules[5],
            If(And(a==True, b==False, c==False), rules[4],
            If(And(a==False, b==True, c==True), rules[3],
            If(And(a==False, b==True, c==False), rules[2],
            If(And(a==False, b==False, c==True), rules[1],
            If(And(a==False, b==False, c==False), rules[0], False))))))))

    S=[[Bool("%d_%d" % (s, i)) for i in range(WIDTH)] for s in range(STATES)]

    s=Solver()

    if WRAPPED==False:
        for st in range(STATES):
           s.add(S[st][0]==False)
           s.add(S[st][WIDTH-1]==False)

    #s.add(S[0][15]==True)

    if WRAPPED==False:
        for st in range(1,STATES):
            for i in range(1, WIDTH-1):
                s.add(S[st][i] == f(S[st-1][i-1], S[st-1][i], S[st-1][i+1]))
    else:
        for st in range(1,STATES):
            for i in range(WIDTH):
                s.add(S[st][i] == f(S[st-1][(i-1) % WIDTH], S[st-1][i], S[st-1][(i+1)
                    % WIDTH]))

    def is_empty(st):
        t=[]
        for i in range(WIDTH):
            t.append(S[st][i]==False)
        return And(*t)

    def is_full(st):
        t=[]
        for i in range(WIDTH):
            t.append(S[st][i]==True)
        return And(*t)

    def non_equal_states (st1, st2):
        t=[]
        for i in range(WIDTH):
            t.append(S[st1][i] != S[st2][i])
        return Or(*t)

    #s.add(non_equal_states(0, 1))

    for st in range(STATES):
        s.add(is_empty(st)==False)
        s.add(is_full(st)==False)

    # first and last states are equal to each other:
    if WRAPPED==False:
        for i in range(1,WIDTH-1):
            if SHIFTED==0:
                s.add(S[0][i]==S[STATES-1][i])
            if SHIFTED==1:
                s.add(S[0][i]==S[STATES-1][i-1])
            if SHIFTED==2:
                s.add(S[0][i]==S[STATES-1][i+1])
    else:
        for i in range(WIDTH):
            if SHIFTED==0:
                s.add(S[0][i]==S[STATES-1][i % WIDTH])
            if SHIFTED==1:
                s.add(S[0][i]==S[STATES-1][(i-1) % WIDTH])
            if SHIFTED==2:
                s.add(S[0][i]==S[STATES-1][(i+1) % WIDTH])

    if s.check()==unsat:
        return
        #print "unsat"
        #exit(0)

    m=s.model()

    print ("RULE=%d STATES=%d, WRAPPED=%s, SHIFTED=%d" % (RULE, STATES, str(WRAPPED),
         SHIFTED))
    for st in range(STATES):
        t=""
        for i in range(WIDTH):
            if str(m[S[st][i]])=="False":
                t=t+"."
            else:
                t=t+"*"
        print (t)

for RULE in range(0, 256):
    for STATES in range(2, 10):
        if True:
        #for WRAPPED in [False, True]:
            WRAPPED=True
            for SHIFTED in [0,1,2]:
                _try (RULE, STATES, WRAPPED, SHIFTED)
```

Some *oscillators* and *gliders* are nice:

```
RULE=26 STATES=7, WRAPPED=True, SHIFTED=1
.***..***..***.
.*.**.*.**.*.**
...**...**...**
*..***..***..**
**.*.**.*.**.*.
**...**...**...
***..***..***..

RULE=29 STATES=3, WRAPPED=True, SHIFTED=1
..*..*..*..*..*
*..*..*..*..*..
.*..*..*..*..*.

RULE=30 STATES=4, WRAPPED=True, SHIFTED=1
**..***..***..*
.**.*.**.*.**.*
****.****.****.
*..***..***..**

RULE=38 STATES=4, WRAPPED=True, SHIFTED=0
*.**.**.**.**.*
**.**.**.**.**.
.**.**.**.**.**
*.**.**.**.**.*

RULE=40 STATES=5, WRAPPED=True, SHIFTED=1
*...**...**...*
.*....*....*...
.**...**...**..
...*....*....*.
...**...**...**

RULE=41 STATES=5, WRAPPED=True, SHIFTED=1
...*....*....*.
...**...**...**
*....*....*....
**...**...**...
..*....*....*..

RULE=42 STATES=3, WRAPPED=True, SHIFTED=2
*..*..*..*..*..
**.**.**.**.**.
.*..*..*..*..*.

RULE=42 STATES=5, WRAPPED=True, SHIFTED=2
.*.*..*.*..*.*.
.*.**.*.**.*.**
.*..*.*..*.*..*
.**.*.**.*.**.*
..*.*..*.*..*.*

RULE=43 STATES=7, WRAPPED=True, SHIFTED=0
**.**.**.**.**.
.*..*..*..*..*.
.**.**.**.**.**
..*..*..*..*..*
```

```
*.**.**.**.**.*
*..*..*..*..*..
**.**.**.**.**.

RULE=44 STATES=3, WRAPPED=True, SHIFTED=1
.**.**.**.**.**
*..*..*..*..*..
**.**.**.**.**.

RULE=44 STATES=5, WRAPPED=True, SHIFTED=1
*...**...**...*
.*....*....*...
.**...**...**..
...*....*....*.
...**...**...**

RULE=45 STATES=3, WRAPPED=True, SHIFTED=1
*..*..*..*..*..
**.**.**.**.**.
..*..*..*..*..*

RULE=60 STATES=4, WRAPPED=True, SHIFTED=1
.*.*..*.*..*.*.
.****.****.****
**...**...**...
*.*..*.*..*.*..

RULE=60 STATES=5, WRAPPED=True, SHIFTED=2
*..*...****.*.*
.*.**..*...****
****.*.**..*...
*...****.*.**..
**..*...****.*.

RULE=72 STATES=3, WRAPPED=True, SHIFTED=0
**..*.*..*.*...
..**...**...*.*
**..*.*..*.*...

RULE=73 STATES=3, WRAPPED=True, SHIFTED=0
.**..*.*..*.*..
*..**...**...*.
.**..*.*..*.*..

RULE=74 STATES=5, WRAPPED=True, SHIFTED=1
..*....*....*..
.*.*..*.*..*.*.
*...**...**...*
**.*.**.*.**.*.
.*....*....*...

RULE=75 STATES=3, WRAPPED=True, SHIFTED=1
..*..*..*..*..*
**.**.**.**.**.
.*..*..*..*..*.

RULE=76 STATES=3, WRAPPED=True, SHIFTED=0
*.**.**.*.*.*..
.*..*..*.*.*.**
*.**.**.*.*.*..
```

```
RULE=78 STATES=3, WRAPPED=True, SHIFTED=1
..**.***..**.**
**.**..***.**.*
.**.***..**.**.

RULE=82 STATES=7, WRAPPED=True, SHIFTED=2
*.**.*.**.*.**.
..**...**...**.
.***..***..***.
**.*.**.*.**.*.
**...**...**...
**..***..***..*
.*.**.*.**.*.**

RULE=90 STATES=4, WRAPPED=True, SHIFTED=0
..*.**..**.*...
.*..******..*..
*.***....***.*.
..*.**..**.*...

RULE=98 STATES=3, WRAPPED=True, SHIFTED=0
*.*.**.*.**.*.*
*.*..*.*..*.*..
*.*.**.*.**.*.*

RULE=100 STATES=3, WRAPPED=True, SHIFTED=2
*.**.**.**.**.*
.*..*..*..*..*.
**.**.**.**.**.

RULE=101 STATES=3, WRAPPED=True, SHIFTED=2
*..*..*..*..*..
*.**.**.**.**.*
.*..*..*..*..*.

RULE=102 STATES=4, WRAPPED=True, SHIFTED=2
***.****.****.*
..**...**...**.
.*.*..*.*..*.*.
****.****.****.

RULE=102 STATES=6, WRAPPED=True, SHIFTED=0
***.*.**..*...*
..****.*.**..*.
.*...****.*.**.
**..*...****.*.
.*.**..*...****
***.*.**..*...*

RULE=105 STATES=4, WRAPPED=True, SHIFTED=0
.**...**...**..
*..*.*..*.*..*.
****.****.****.
.**...**...**..

RULE=105 STATES=6, WRAPPED=True, SHIFTED=0
.*.*...**..**..
**.**.*..**..*.
......***..***.
.....*.*.**.*.*
*...**.*....*.*
```

```
.*.*...**..**..

RULE=105 STATES=7, WRAPPED=True, SHIFTED=0
*....*....*....
**..***..***..*
*.**.*.**.*.**.
*....*....*....
**..***..***..*
*.**.*.**.*.**.
*....*....*....

RULE=106 STATES=3, WRAPPED=True, SHIFTED=1
***..***..***..
..***..***..***
**..***..***..*

RULE=106 STATES=6, WRAPPED=True, SHIFTED=0
***..*.**.*...*
..****..*.**.*.
.*...****..*.**
.**.*...****..*
..*.**.*...****
***..*.**.*...*

RULE=106 STATES=8, WRAPPED=True, SHIFTED=0
.*.**...**..**.
**..**.*.***.**
.***.*.*...*...
*..*.*.**.***..
****.*..*...***
...*.*****.*...
..**.....*.**..
.*.**...**..**.

RULE=108 STATES=5, WRAPPED=True, SHIFTED=0
*****.***.***.*
.....*...*...*.
....***.***.***
*..*...*...*...
*****.***.***.*

RULE=108 STATES=9, WRAPPED=True, SHIFTED=0
.*..*...*.*....
******.*****...
......*.....*.*
*....***...****
.*..*...*.*....
******.*****...
......*.....*.*
*....***...****
.*..*...*.*....

RULE=109 STATES=3, WRAPPED=True, SHIFTED=0
***.*.**..***..
.*.***..**.*.**
***.*.**..***..

RULE=110 STATES=3, WRAPPED=True, SHIFTED=1
**.**.**..***.*
.**.**.***..**.
*.**.**..***.**
```

```
RULE=110 STATES=4, WRAPPED=True, SHIFTED=0
*...**.****...*
**.*.**...**.*.
.****.**.*.****
*...**.****...*

RULE=110 STATES=7, WRAPPED=True, SHIFTED=0
*.*.**...**...*
****.**.*.**.*.
...**.****.****
*.*.**...**...*
****.**.*.**.*.
...**.****.****
*.*.**...**...*

RULE=120 STATES=8, WRAPPED=True, SHIFTED=0
*.**.*.*..*..**
..*..*.*******.
.*****.*......*
.*.....**....**
.**...**.*..**.
**.*.**..****.*
...*.*.***....*
*.**.*.*..*..**

RULE=122 STATES=7, WRAPPED=True, SHIFTED=0
*****.**.**.*.*
....*.**.**.*.*
*..**.**.**.*.*
*****.**.**.*.*
....*.**.**.*.*
*..**.**.**.*.*
*****.**.**.*.*

RULE=124 STATES=4, WRAPPED=True, SHIFTED=0
.**....*..*****
**.*..*****....
*.*****....*..*
.**....*..*****

RULE=124 STATES=7, WRAPPED=True, SHIFTED=0
..**.*****....*
***.**....*..**
...**.*..*****.
..**.*****....*
***.**....*..**
...**.*..*****.
..**.*****....*

RULE=128 STATES=3, WRAPPED=True, SHIFTED=0
...*...*....***
.*...*...**....
...*...*....***

RULE=129 STATES=3, WRAPPED=True, SHIFTED=0
..*....**....*.
*...**....**...
..*....**....*.

RULE=134 STATES=4, WRAPPED=True, SHIFTED=1
```

```
...*....*....*.
**...**...**...
.*.*..*.*..*.*.
..*....*....*..

RULE=135 STATES=4, WRAPPED=True, SHIFTED=1
..**...**...**.
*..*.*..*.*..*.
....*....*....*
.**...**...**..

RULE=137 STATES=4, WRAPPED=True, SHIFTED=0
*.....*..*..*..
.****..*..*..*.
..**.*..*..*..*
*.....*..*..*..

RULE=148 STATES=4, WRAPPED=True, SHIFTED=2
.*.*..*.*..*.*.
..*....*....*..
*...**...**...*
..*.*..*.*..*.*

RULE=149 STATES=4, WRAPPED=True, SHIFTED=2
*...**...**...*
..*.*..*.*..*.*
...*....*....*.
**...**...**...

RULE=150 STATES=7, WRAPPED=True, SHIFTED=0
**.****.****.**
.***..***..***.
.*.*..*.*..*.*.
..*....*....*..
*...**...**...*
*.*.**.*.**.*.*
**.****.****.**

RULE=154 STATES=7, WRAPPED=True, SHIFTED=1
.*.**.*.**.*.**
...**...**...**
**.****.****.**
.*.*..*.*..*.*.
....*....*....*
***..***..***..
*.**.*.**.*.**.

RULE=166 STATES=7, WRAPPED=True, SHIFTED=1
*.*.**.*.**.*.*
****.****.****.
...**...**...**
.*..*.*..*.*..*
**..***..***..*
.*....*....*...
.*.**.*.**.*.**

RULE=167 STATES=7, WRAPPED=True, SHIFTED=1
*.*..*.*..*.*..
***..***..***..
.**...**...**..
..*.*..*.*..*.*
```

```
..***..***..***
...**...**...**
.*..*.*..*.*..*

RULE=169 STATES=6, WRAPPED=True, SHIFTED=0
..**.*..*.***..
*....**.*..*.**
.***....**.*..*
..*.***....**.*
*.*..*.***....*
..**.*..*.***..

RULE=169 STATES=8, WRAPPED=True, SHIFTED=0
.*.*.***.****..
.*.*..*...**.**
.*.**.***......
.*.....*.******
.*****.*..****.
..***..**..**.*
*..*.*...*....*
.*.*.***.****..

RULE=182 STATES=6, WRAPPED=True, SHIFTED=0
*..*.*****.*..*
*..***...***..*
*..*.*.*.*.*..*
*..*********..*
*..*.......*..*
*..*.*****.*..*

RULE=188 STATES=7, WRAPPED=True, SHIFTED=2
.**.*.**.*.**.*
**.****.****.**
..**...**...**.
*.*.**.*.**.*.*
.****.****.****
**...**...**...
*.**.*.**.*.**.

RULE=193 STATES=4, WRAPPED=True, SHIFTED=0
***.**..****..*
**.....*.**..*.
...****.....*..
***.**..****..*

RULE=195 STATES=5, WRAPPED=True, SHIFTED=2
*.***....*.*..*
*..**.***....*.
..*.*..**.***..
**....*.*..**.*
**.***....*.*..

RULE=195 STATES=6, WRAPPED=True, SHIFTED=0
*..*.*....***.*
*.*....***.**..
....***.**..*.*
.***.**..*.*...
*.**..*.*....**
*..*.*....***.*

RULE=201 STATES=5, WRAPPED=True, SHIFTED=0
```

```
......***...***
******.*.***.*.
.****.....*....
*.**.*****.****
......***...***

RULE=201 STATES=9, WRAPPED=True, SHIFTED=0
..****.....*...
**.**.*****.***
*......***...**
.******.*.***.*
..****.....*...
**.**.*****.***
*......***...**
.******.*.***.*
..****.....*...

RULE=210 STATES=7, WRAPPED=True, SHIFTED=2
*....*....*....
..***..***..***
.**.*.**.*.**.*
.**...**...**..
***.****.****.*
..*.*..*.*..*.*
.*....*....*...

RULE=225 STATES=8, WRAPPED=True, SHIFTED=0
.**.*.......***
....*.******.*.
*****..****..*.
.***..*.**..**.
*.*..**....*...
*.*.*...****.**
..*.*.**.**...*
.**.*.......***
```

`SHIFTED=0` means *oscillator*, `SHIFTED=1` means *glider* slipping left, `SHIFTED=2` — slipping right.

Chapter 22

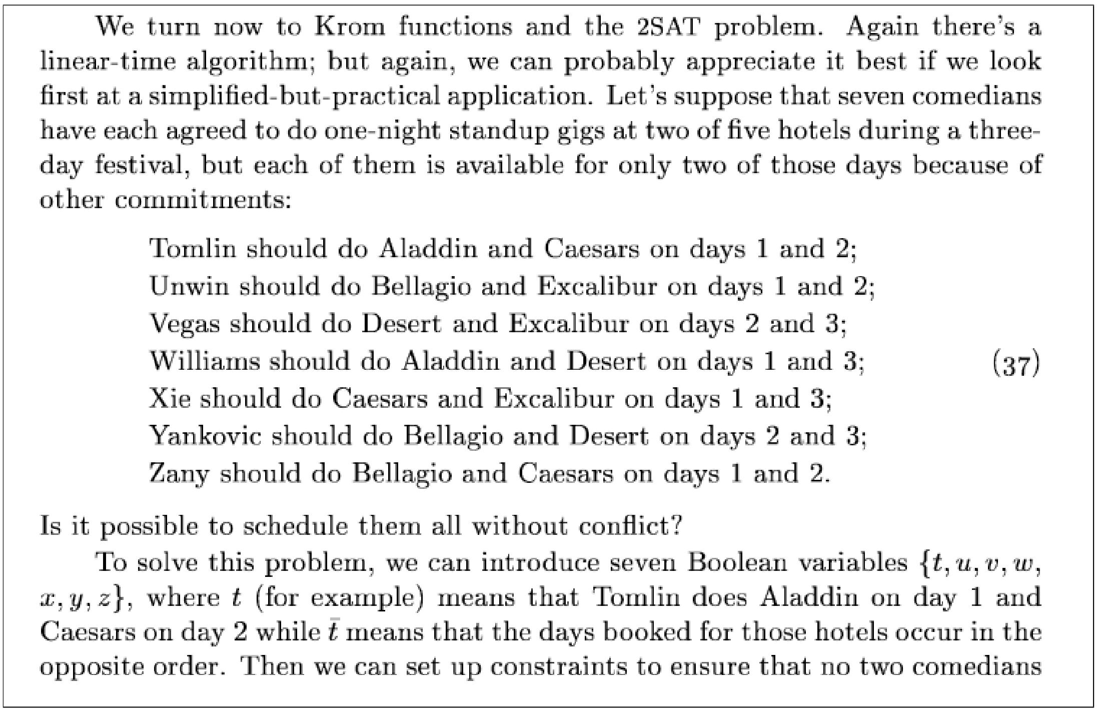
Everything else

22.1 Minimal Unsatisfiable Subformula

It's a bit frustrating to get a depressing 'UNSAT' message from a SAT solver, and nothing else. What can you do for diagnostics? What can you do to fix your SAT instance? What can you do to unclog your (overconstrained) instance?

In some other place of this book, I used UNSAT core generation in Z3 to get a list of conflicting cells in spreadsheet 3.14.2. Here I'll see what can be done for a SAT instance.

I'll use this problem from D.Knuth's TAOCP 7.1.1[1]:



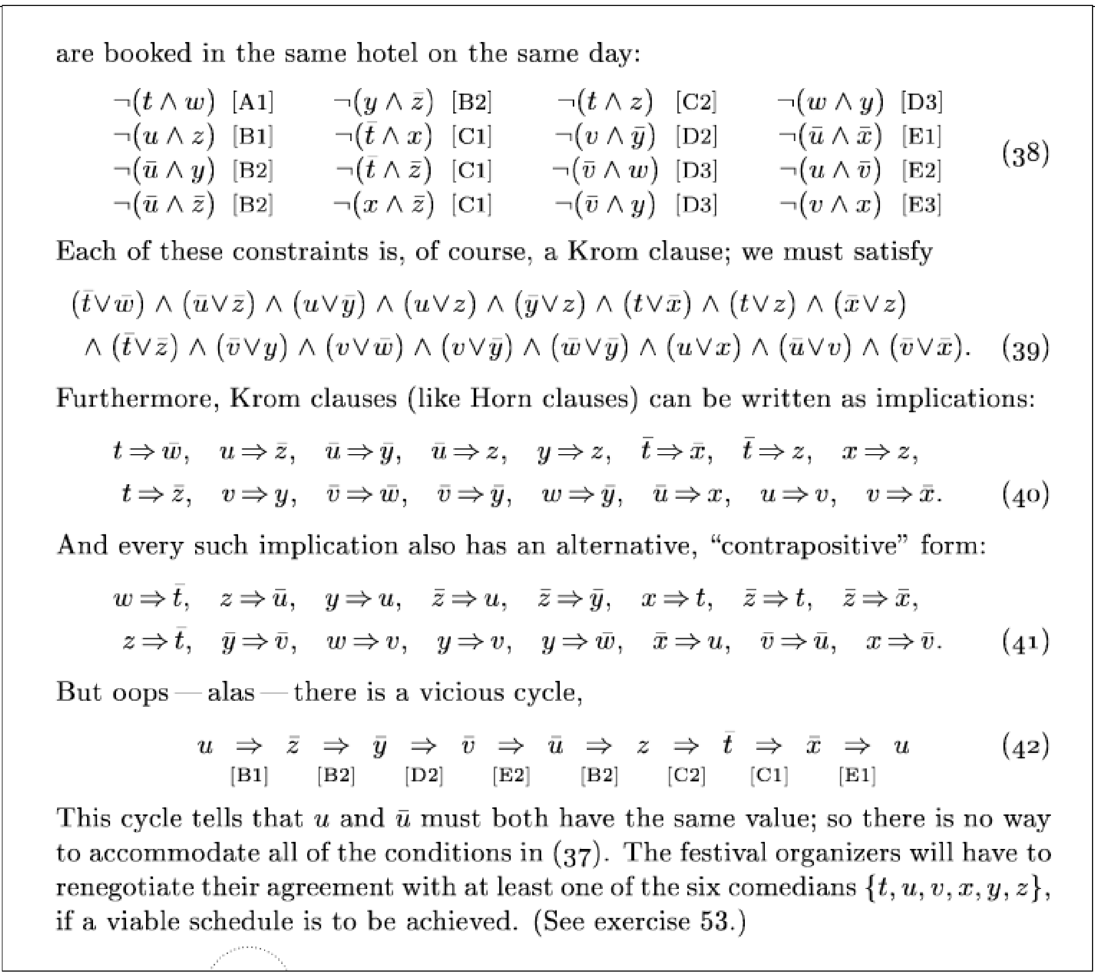
We turn now to Krom functions and the 2SAT problem. Again there's a linear-time algorithm; but again, we can probably appreciate it best if we look first at a simplified-but-practical application. Let's suppose that seven comedians have each agreed to do one-night standup gigs at two of five hotels during a three-day festival, but each of them is available for only two of those days because of other commitments:

Tomlin should do Aladdin and Caesars on days 1 and 2;
Unwin should do Bellagio and Excalibur on days 1 and 2;
Vegas should do Desert and Excalibur on days 2 and 3;
Williams should do Aladdin and Desert on days 1 and 3; (37)
Xie should do Caesars and Excalibur on days 1 and 3;
Yankovic should do Bellagio and Desert on days 2 and 3;
Zany should do Bellagio and Caesars on days 1 and 2.

Is it possible to schedule them all without conflict?

To solve this problem, we can introduce seven Boolean variables $\{t, u, v, w, x, y, z\}$, where t (for example) means that Tomlin does Aladdin on day 1 and Caesars on day 2 while $\bar{t}$ means that the days booked for those hotels occur in the opposite order. Then we can set up constraints to ensure that no two comedians

[1] http://www.cs.utsa.edu/~wagner/knuth/fasc0b.pdf#page=18&zoom=200,-6,389



are booked in the same hotel on the same day:

$$\begin{array}{llllllll} \neg(t\wedge w) & [\mathrm{A1}] & \neg(y\wedge\bar z) & [\mathrm{B2}] & \neg(t\wedge z) & [\mathrm{C2}] & \neg(w\wedge y) & [\mathrm{D3}] \\ \neg(u\wedge z) & [\mathrm{B1}] & \neg(\bar t\wedge x) & [\mathrm{C1}] & \neg(v\wedge\bar y) & [\mathrm{D2}] & \neg(\bar u\wedge\bar x) & [\mathrm{E1}] \\ \neg(\bar u\wedge y) & [\mathrm{B2}] & \neg(\bar t\wedge\bar z) & [\mathrm{C1}] & \neg(\bar v\wedge w) & [\mathrm{D3}] & \neg(u\wedge\bar v) & [\mathrm{E2}] \\ \neg(\bar u\wedge\bar z) & [\mathrm{B2}] & \neg(x\wedge\bar z) & [\mathrm{C1}] & \neg(\bar v\wedge y) & [\mathrm{D3}] & \neg(v\wedge x) & [\mathrm{E3}] \end{array} \qquad (38)$$

Each of these constraints is, of course, a Krom clause; we must satisfy

$$(\bar t\vee\bar w)\wedge(\bar u\vee\bar z)\wedge(u\vee\bar y)\wedge(u\vee z)\wedge(\bar y\vee z)\wedge(t\vee\bar x)\wedge(t\vee z)\wedge(\bar x\vee z)$$
$$\wedge(\bar t\vee\bar z)\wedge(\bar v\vee y)\wedge(v\vee\bar w)\wedge(v\vee\bar y)\wedge(\bar w\vee\bar y)\wedge(u\vee x)\wedge(\bar u\vee v)\wedge(\bar v\vee\bar x). \qquad (39)$$

Furthermore, Krom clauses (like Horn clauses) can be written as implications:

$$t\Rightarrow\bar w,\quad u\Rightarrow\bar z,\quad \bar u\Rightarrow\bar y,\quad \bar u\Rightarrow z,\quad y\Rightarrow z,\quad \bar t\Rightarrow\bar x,\quad \bar t\Rightarrow z,\quad x\Rightarrow z,$$
$$t\Rightarrow\bar z,\quad v\Rightarrow y,\quad \bar v\Rightarrow\bar w,\quad \bar v\Rightarrow\bar y,\quad w\Rightarrow\bar y,\quad \bar u\Rightarrow x,\quad u\Rightarrow v,\quad v\Rightarrow\bar x. \qquad (40)$$

And every such implication also has an alternative, “contrapositive” form:

$$w\Rightarrow\bar t,\quad z\Rightarrow\bar u,\quad y\Rightarrow u,\quad \bar z\Rightarrow u,\quad \bar z\Rightarrow\bar y,\quad x\Rightarrow t,\quad \bar z\Rightarrow t,\quad \bar z\Rightarrow\bar x,$$
$$z\Rightarrow\bar t,\quad \bar y\Rightarrow\bar v,\quad w\Rightarrow v,\quad y\Rightarrow v,\quad y\Rightarrow\bar w,\quad \bar x\Rightarrow u,\quad \bar v\Rightarrow\bar u,\quad x\Rightarrow\bar v. \qquad (41)$$

But oops — alas — there is a vicious cycle,

$$u \underset{[\mathrm{B1}]}{\Rightarrow} \bar z \underset{[\mathrm{B2}]}{\Rightarrow} \bar y \underset{[\mathrm{D2}]}{\Rightarrow} \bar v \underset{[\mathrm{E2}]}{\Rightarrow} \bar u \underset{[\mathrm{B2}]}{\Rightarrow} z \underset{[\mathrm{C2}]}{\Rightarrow} \bar t \underset{[\mathrm{C1}]}{\Rightarrow} \bar x \underset{[\mathrm{E1}]}{\Rightarrow} u \qquad (42)$$

This cycle tells that u and $\bar u$ must both have the same value; so there is no way to accommodate all of the conditions in (37). The festival organizers will have to renegotiate their agreement with at least one of the six comedians $\{t, u, v, x, y, z\}$, if a viable schedule is to be achieved. (See exercise 53.)

I’m assigning 1..7 SAT variables to t,u,v,w,x,y,z variables and translating this to a CNF file (lines prefixed with ”c ” are comments, of course):

```
p cnf 7 16
c variables:
c 1     t
c 2     u
c 3     v
c 4     w
c 5     x
c 6     y
c 7     z
c clauses.
c first line (commented) is clause number and variable names.
c second line: CNF clause
c #1 -t -w
-1 -4 0
c #2 -u -z
-2 -7 0
c #3 u y
2 -6 0
c #4 u z
2 7 0
c #5 -y z
-6 7 0
c #6 t -x
1 -5 0
c #7 t z
1 7 0
c #8 -x z
```

```
-5 7 0
c #9 -t -z
-1 -7 0
c #10 -v y
-3 6 0
c #11 v -w
3 -4 0
c #12 v -y
3 -6 0
c #13 -w -y
-4 -6 0
c #14 y x
2 5 0
c #15 -u v
-2 3 0
c #16 -v -x
-3 -5 0
```

The instance is UNSAT, of course. Now I'm running PicoMUS utility from Picosat SAT solver, that extracts MUS[2]:

```
c [picomus] WARNING: no output file given
c [picomus] WARNING: PicoSAT compiled without trace generation
c [picomus] WARNING: core extraction disabled
s UNSATISFIABLE
c [picomus] computed MUS of size 8 out of 16 (50%)
v 2
v 4
v 5
v 6
v 9
v 10
v 14
v 15
v 0
```

This is a list of clauses. If all of them are removed from your instances, it will be transformed from UNSAT to SAT. Let's see, what are these clauses?

```
-2 -7 0 # 2     -u -z
2 7 0   # 4     u z
-6 7 0  # 5     -y z
1 -5 0  # 6     t -x
-1 -7 0 # 9     -t -z
-3 6 0  # 10    -v y
-2 3 0  # 15    -u v
```

These are indeed clauses with variables that are in the vicious cycle[3] according to D.Knuth.

Note that the MUS is not the smallest possible (sub)set, but just a set that you can use for diagnostics. *Minimal* in MUS means that is cannot be reduced in its form. However, smallest MUSes are still possible to exist, but PicoMUS isn't guaranteed to find a smallest MUS.

Now there is another tool in Picosat: PicoMCS, that extracts MCS[4]. What can it say?

```
s UNSATISFIABLE
v 10 0
v 14 0
v 15 0
v 2 0
v 5 0
v 6 0
```

[2] Minimal Unsatisfiable Subformula

[3] https://en.wikipedia.org/wiki/Virtuous_circle_and_vicious_circle

[4] Minimal Correction Subset

```
v 8 4 0
v 9 0
```

Each line in list is a list of clauses. If a list of clauses is removed from your instances, it transforms from UNSAT to SAT. Indeed, 10th clause is an element of *vicious cycle*. And so is 15th, etc. But 14th clause is not. But removing it from the instance will solve the problem. Also, if 8th and 4th clauses are both to be removed from the instance, it will be SAT, but not one-by-one.

Why not always using PicoMCS instead of PicoMUS? It's way slower. And in practice, you can be satisfied with larger MUS.

Now what if we can represent each clause as weighted and use MaxSAT solver? (The same Open-WBO, I use so often.)

I prepend each clause with '1', that is a (smallest) weight of each clause:

```
p wcnf 7 16 1000
1 -1 -4 0
1 -2 -7 0
1 2 -6 0
1 2 7 0
1 -6 7 0
1 1 -5 0
1 7 0
1 -5 7 0
1 -1 -7 0
1 -3 6 0
1 3 -4 0
1 3 -6 0
1 -4 -6 0
1 2 5 0
1 -2 3 0
1 -3 -5 0
```

... and asking Open-WBO MaxSAT solver to find such an assignment, that can satisfy as many clauses, as possible:

```
s OPTIMUM FOUND
v -1 2 3 -4 -5 6 7
```

Let's see, which clauses gets satisfied with this assignment? I manually added comments to the WCNF instance:

```
p wcnf 7 16 1000
1 -1 -4 0       # -t -w         OK
1 -2 -7 0       # -u -z         UNSAT
1 2 -6 0        # u y           OK
1 2 7 0         # u z           OK
1 -6 7 0        # -y z          OK
1 1 -5 0        # t -x          OK
1 1 7 0         # t z           OK
1 -5 7 0        # -x z          OK
1 -1 -7 0       # -t -z         OK
1 -3 6 0        # -v y          OK
1 3 -4 0        # v -w          OK
1 3 -6 0        # v -y          OK
1 -4 -6 0       # -w -y         OK
1 2 5 0         # y x           OK
1 -2 3 0        # -u v          OK
1 -3 -5 0       # -v -x         OK
```

The only 2nd clause is to be removed to break the *vicious cycle*.

Again, MaxSAT solvers are way slower than MUS extractors. So they cannot be always used in practice.

22.1.1 MUSer2

This is a tool[5] acting as a front-end MUS extractor on top of a SAT-solver like MiniSat, PicoSat, Glucose, UBCSAT, etc. Unlike PicoMUS, it produces a CNF file for our example, but it contains 8 clauses as well:

[5] https://github.com/meelgroup/muser

```
c 0 unknown clauses, 8 necessary clauses.
p cnf 7 8
-2 -7 0
2 7 0
-6 7 0
1 -5 0
-1 -7 0
-3 6 0
2 5 0
-2 3 0
```

22.2 SAT solver on top of regex matcher

A SAT problem is an NP-problem, while regex matching is not. However, a quite popular regex *backreferences* extension extends regex matching to a (hard) NP-problem. Backreferences usually denoted as `\1`, `\2`, etc.

Perhaps, the most practical use of backreferences I've heard is HTML tag matching[6]

Listing 22.1: This regex isn't escaped properly

```
<(.*)> ... </\1>
```

To match successfully, the second group must coincide with the first, like `<b>test</b>`, but not `<a>test</b>`. Another practical usage I've heard: match `"string"` or `'string'`, but not `"string'`.

Also, you can find two longest repeating substrings in an input string [7].

Some other uses are very arcane: divide input number by $\sqrt{2}$[8], detecting factorial number[9].

This text has been inspired by "Reduction of 3-CNF-SAT to Perl Regular Expression Matching"[10], please read it first. However the author incorrectly states that only 3SAT problems are solvable. In fact, any SAT instance is solvable, consisting of clauses of arbitrary sizes.

Also, since my SAT/CNF instances usually has more variables than 9 and I can't use backreferences like `\1 ... \9`, I use different method (Python):

```
The syntax for backreferences in an expression such as (...)\1 refers to the number
    of the group. 'Theres naturally a variant that uses the group name instead of the
    number. This is another Python extension: (?P=name) indicates that the contents of
     the group called name should again be matched at the current point. The regular
    expression for finding doubled words, \b(\w+)\s+\1\b can also be written as \b(?P<
    word>\w+)\s+(?P=word)\b
```

(https://docs.python.org/3/howto/regex.html)

See also: https://www.regular-expressions.info/replacebackref.html.

Now let's take this small CNF instance:

```
p cnf 5 11
-2 -5 0
-2 -4 0
-4 -5 0
-2 -3 0
-1 -4 0
-1 -5 0
-1 -2 0
-1 -3 0
-3 -4 0
-3 -5 0
1 2 3 4 5 0
```

This is what I call *popcnt1*: only 1 variable must be true, all the rest are always false.

[6] https://www.regular-expressions.info/backref.html

[7] https://stackoverflow.com/questions/9177647/regular-epxressions-that-matches-the-longest-repeating-sequence https://stackoverflow.com/questions/44465284/match-longest-repeating-sequence-that-is-not-made-of-a-repeating-sequence

[8] https://codegolf.stackexchange.com/questions/198427/shift-right-by-half-a-bit/198428#198428

[9] https://codegolf.stackexchange.com/questions/121731/is-this-number-a-factorial/178979#178979

[10] https://perl.plover.com/NPC/NPC-3SAT.html

```
% picosat --all popcnt1.cnf

s SATISFIABLE
v -1 -2 -3 -4 5 0
s SATISFIABLE
v -1 -2 -3 4 -5 0
s SATISFIABLE
v -1 -2 3 -4 -5 0
s SATISFIABLE
v -1 2 -3 -4 -5 0
s SATISFIABLE
v 1 -2 -3 -4 -5 0
s SOLUTIONS 5
```

Now let's translate it to regex:

```
string=xxxxx;x,x,x,x,x,x,x,x,x,x,x,
pattern=^(?P<a_1>x?)(?P<a_2>x?)(?P<a_3>x?)(?P<a_4>x?)(?P<a_5>x?).*;(?:(?P=a_2)x|(?P=
    a_5)x),(?:(?P=a_2)x|(?P=a_4)x),(?:(?P=a_4)x|(?P=a_5)x),(?:(?P=a_2)x|(?P=a_3)x)
    ,(?:(?P=a_1)x|(?P=a_4)x),(?:(?P=a_1)x|(?P=a_5)x),(?:(?P=a_1)x|(?P=a_2)x),(?:(?P=
    a_1)x|(?P=a_3)x),(?:(?P=a_3)x|(?P=a_4)x),(?:(?P=a_3)x|(?P=a_5)x),(?:(?P=a_1)|(?P=
    a_2)|(?P=a_3)|(?P=a_4)|(?P=a_5)),
SAT
1 -2 -3 -4 -5
```

You see: an input string has as many x's as many variables the CNF instance has. Then, as many `x,` substrings, as many clauses the instance has.

Take the first part of the pattern: `^(?P<a_1>x?)(?P<a_2>x?)(?P<a_3>x?)(?P<a_4>x?)(?P<a_5>x?).*; ...`

Roughly, it means that the group can match `x`, or may not. Let matcher decide on its own. The second part of the pattern are clauses. For the `1 2 3 4 5` clause we have `(?:(?P=a_1)|(?P=a_2)|(?P=a_3)|(?P=a_4)|(?P=a_5))`, there. That means, that the whole group *must* match `x`, but we don't tell how: one of 5 subgroups may match, and each subgroup is actually backreference to the first part of pattern. But these terms are all positive. What about negative terms?

For the `-2 -5` clause, we have `(?:(?P=a_2)x|(?P=a_5)x)`, there. That means that the whole group *must* match `x`, and again, we don't tell how, but both backreferences `a_2` and `a_5` are prohibited, if present simultaneously. It's OK for `a_2` to match `x`, but then the second part of the union would match. It's OK for `a_5` to match `x`, and then the first part of the union would match. Also, it's OK for both `a_2` and `a_5` match nothing: either part of union will match `x` then.

If regex can't match, this means the input instance is unsatisfiable.

Likewise, I can run my *popcnt4* instance, which commands to be 4 of input 8 variables be true:

```
string=xxxxxxxx;x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x
    ,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x
    ,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,
pattern=^(?P<a_1>x?)(?P<a_2>x?)(?P<a_3>x?)(?P<a_4>x?)(?P<a_5>x?)(?P<a_6>x?)(?P<a_7>x
    ?)(?P<a_8>x?).*;(?:(?P=a_1)x|(?P=a_2)x|(?P=a_3)x|(?P=a_4)x|(?P=a_5)x),(?:(?P=a_1)x
    |(?P=a_2)x|(?P=a_3)x|(?P=a_4)x|(?P=a_6)x),(?:(?P=a_1)x|(?P=a_2)x|(?P=a_3)x|(?P=a_4
    )x|(?P=a_7)x),(?:(?P=a_1)x|(?P=a_2)x|(?P=a_3)x|(?P=a_4)x|(?P=a_8)x),(?:(?P=a_1)x
    |(?P=a_2)x|(?P=a_3)x|(?P=a_5)x|(?P=a_6)x),(?:(?P=a_1)x|(?P=a_2)x|(?P=a_3)x|(?P=a_5
    )x|(?P=

...

P=a_7)|(?P=a_8)),(?:(?P=a_3)|(?P=a_5)|(?P=a_6)|(?P=a_7)|(?P=a_8)),(?:(?P=a_4)x|(?P=
    a_5)x|(?P=a_6)x|(?P=a_7)x|(?P=a_8)x),(?:(?P=a_4)|(?P=a_5)|(?P=a_6)|(?P=a_7)|(?P=
    a_8)),
SAT
1 2 3 4 -5 -6 -7 -8
```

Also, I managed to solve the *Fred puzzle*8.12 in SAT/CNF form:

```
string=xxxxxxxxxxxxxxxxxxxxxxxxxxxx;x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x
    ,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,
pattern=^(?P<a_1>x?)(?P<a_2>x?)(?P<a_3>x?)(?P<a_4>x?)(?P<a_5>x?)(?P<a_6>x?)(?P<a_7>x
    ?)(?P<a_8>x?)(?P<a_9>x?)(?P<a_10>x?)(?P<a_11>x?)(?P<a_12>x?)(?P<a_13>x?)(?P<a_14>x
    ?)(?P<a_15>x?)(?P<a_16>x?)(?P<a_17>x?)(?P<a_18>x?)(?P<a_19>x?)(?P<a_20>x?)(?P<a_21
    >x?)(?P<a_22>x?)(?P<a_23>x?)(?P<a_24>x?)(?P<a_25>x?)(?P<a_26>x?)(?P<a_27>x?)(?P<
    a_28>x?).*;(?:(?P=a_1)x),(?:(?P=a_2)),(?:(?P=a_3)x|(?P=a_4)x|(?P=a_9)x),(?:(?P=a_3
    )|(?P=a_4)|(?P=a_9)x),(?:(?P=a_3)|(?P=a_4)x|(?P=a_9)),(?:(?P=a_3)x|(?P=a_4)|(?P=
    a_9)),(?:(?P=a_10)x|(?P=a_9)x),(?:(?P=a_10)|(?P=a_9)),(?:(?P=a_11)x|(?P=a_10)x)
    ,(?:(?P=a_11)|(?P=a_10)),(?:(?P=a_11)),(?:(?P=a_5)x|(?P=a_6)x|(?P=a_12)x),(?:(?P=
    a_5)|(?P=a_6)|(?P=a_12)x),(?:(?P=a_5)|(?P=a_6)x|(?P=a_12)),(?:(?P=a_5)x|(?P=a_6)
    |(?P=a_12)),(?:(?P=a_13)x|(?P=a_12)x),(?:(?P=a_13)|(?P=a_12)),(?:(?P=a_14)x|(?P=
    a_13)x),(?:(?P=a_14)|(?P=a_13)),(?:(?P=a_14)),(?:(?P=a_7)x|(?P=a_8)x|(?P=a_15)x)
    ,(?:(?P=a_7)|(?P=a_8)|(?P=a_15)x),(?:(?P=a_7)|(?P=a_8)x|(?P=a_15)),(?:(?P=a_7)x|(?
    P=a_8)|(?P=a_15)),(?:(?P=a_16)x|(?P=a_15)x),(?:(?P=a_16)|(?P=a_15)),(?:(?P=a_17)x
    |(?P=a_16)x),(?:(?P=a_17)|(?P=a_16)),(?:(?P=a_17)),(?:(?P=a_3)x|(?P=a_6)x|(?P=a_18
    )),(?:(?P=a_3)|(?P=a_18)x),(?:(?P=a_6)|(?P=a_18)x),(?:(?P=a_8)x|(?P=a_18)x|(?P=
    a_19)x),(?:(?P=a_8)|(?P=a_18)|(?P=a_19)x),(?:(?P=a_8)|(?P=a_18)x|(?P=a_19)),(?:(?P
    =a_8)x|(?P=a_18)|(?P=a_19)),(?:(?P=a_20)x|(?P=a_19)x),(?:(?P=a_20)|(?P=a_19))
    ,(?:(?P=a_20)),(?:(?P=a_21)x|(?P=a_7)x),(?:(?P=a_21)|(?P=a_7)),(?:(?P=a_21)|(?P=
    a_5)|(?P=a_22)x),(?:(?P=a_21)x|(?P=a_22)),(?:(?P=a_5)x|(?P=a_22)),(?:(?P=a_4)x|(?P
    =a_22)x|(?P=a_23)x),(?:(?P=a_4)|(?P=a_22)|(?P=a_23)x),(?:(?P=a_4)|(?P=a_22)x|(?P=
    a_23)),(?:(?P=a_4)x|(?P=a_22)|(?P=a_23)),(?:(?P=a_24)x|(?P=a_23)x),(?:(?P=a_24)|(?
    P=a_23)),(?:(?P=a_24)),(?:(?P=a_3)|(?P=a_7)|(?P=a_25)x),(?:(?P=a_3)x|(?P=a_25))
    ,(?:(?P=a_7)x|(?P=a_25)),(?:(?P=a_6)x|(?P=a_25)x|(?P=a_26)),(?:(?P=a_6)|(?P=a_26)x
    ),(?:(?P=a_25)|(?P=a_26)x),(?:(?P=a_6)x|(?P=a_26)x|(?P=a_27)x),(?:(?P=a_6)|(?P=
    a_26)|(?P=a_27)x),(?:(?P=a_6)|(?P=a_26)x|(?P=a_27)),(?:(?P=a_6)x|(?P=a_26)|(?P=
    a_27)),(?:(?P=a_28)x|(?P=a_27)x),(?:(?P=a_28)|(?P=a_27)),(?:(?P=a_28)),
SAT
-1 2 -3 4 5 -6 7 -8 9 -10 11 12 -13 14 15 -16 17 -18 -19 20 -21 22 -23 24 25 -26 -27
    28
```

It took 3 minutes on my old CPU clocked at 2GHz.

The files, including the *solver* in Python 3: https://smt.st/current_tree/other/regex_SAT/files/.

Of course, all this stuff isn't practical at all. But it demonstrates *reduction* from one problem (regex matching with backreferences) to another (SAT). Find a better algorithm for any of these problem and this would lead to revolution in computer science.

A discussion at HN: https://news.ycombinator.com/item?id=23597573.

22.2.1 Integer factorization using regex (with backreferences)

gbacon at HN pointed to a method of integer factorization using regex.

(Unary encoding is "" for 0, "1" for 1, "11" for 2, "11111" for 5, etc.)

I simplified it a bit, because the first part of `^1?$|^(11+?)\1+$` is just a check against an "1" string (which is prime) and empty string (which is for 0) (`^1?$`), and I removed it for clarity:

```
#!/usr/bin/env python3

import re

#n=12300 # composite
#n=123001 # prime, 27s
#n=12300200 # composite
n=123023 # composite, one factor: 43
#n=123027 # composite, one factor: 3
#n=223099 # prime, 87s

regex=re.compile("^(11+?)\\1+$")
res=regex.match("1"*n)
if res==None:
    print ("prime")
else:
    x=len(res[1])
    y=n/x
    print ("composite: %d * %d = %d" % (x, y, n))
```

It can find factors for small numbers. And here is how it works. In plain English, we asking regex matcher to find such a string, that consists of some number (≥ 2) of ”1”’s (`(11+?)`), which is glueled with the same string (`\1`) arbitrary number of times (`+`).

Of course it’s extremely slow, and even worse than bruteforce. For 87 seconds on my old 2GHz CPU I found out that 223099 is a prime.

But again, this is like a thought experiment. A reduction from one problem (integer factorization) to another (find equal substrings in a string).

22.2.2 Allan Wirth’s version

```
From: Allan Wirth <allanlw(a)gmail.com>
Subject: PCRE JITing your regex sat solver

Hi Dennis,

I really appreciated your simple "regex" to 3-sat reduction.

I was curious to see how much perf could be gained from using the PCRE
JIT with your regex SAT solver, so I reimplemented it in PHP (I don't
know Perl and PHP has native PCRE support).

Source: https://gist.github.com/allanlw/69df509519335b88db886d48503a0f15

Time comparison below. It's about 50% constant speedup. Thought you
might be interested.

Cheers,
Allan
```

22.2.3 One more usage of backreferences

This time, I searched for good words that can serve as examples for my example about Knuth-Morris-Pratt algorithm 5.3. I wanted a list of words that have repeated prefixes and suffixes.

I took a good collection of English words here.

Then I used `sed` to find words with repeated prefixes:

```
sed -E -n '/^(.+)\1(.+)$/p' words_alpha.txt
```

Some of them:

```
eel
oops
ooze
cocoa
cocos
kokos
mimic
cocoon
```

I couldn’t manage `sed` to find repeated suffixes, so I wrote a Racket program to do that (each suffix must have at least two characters):

```
#lang racket

;(define r (pregexp "^(.+)\\1(.+)$")) ; two prefixes
(define r (pregexp "^(.+)(..+)\\2$")) ; two suffixes >=2

(define (f s)
  (regexp-match r s))

```

```
(define result
  (sort
   (filter f (file->lines "words_alpha.txt"))
   (lambda (x y)
     (< (string-length x) (string-length y)))))

(for ([i result])
  (displayln i))
```

Some of these:

```
ceded
crisis
rococo
cantata
```

That sounds as a list of diseases:

```
hydrofluosilicic
integropallialia
interjaculateded
panmyelophthisis
plasmaphoresisis
pneumonophthisis
antihemagglutinin
ophthalmophthisis
bacterioagglutinin
erythrocytoschisis
phytohemagglutinin
thoracoceloschisis
craniorhachischisis
phytohaemagglutinin
thoracogastroschisis
```

I couln’d stand the itch and tried to find all words with 3 repeated suffixes:

```
(define r (pregexp "^(.+)(.+)\\2\\2$"))
```

That includes both words with 3 repeated characters at the end and the rare term ’ratatat’ – thricely repeated ’at’ suffix:

```
brrr
ieee
mmmm
oooo
viii
xiii
xviii
xxiii
ratatat
earlesss
```

(’earlesss’ seems to be a typo in the list of words I used.)

22.3 Ménage problem

```
In combinatorial mathematics, the ménage problem or problème des ménages[1] asks for
   the number of different ways in which it is possible to seat a set of male-female
   couples at a dining table so that men and women alternate and nobody sits next to
   his or her partner. This problem was formulated in 1891 by Édouard Lucas and
   independently, a few years earlier, by Peter Guthrie Tait in connection with knot
   theory.[2] For a number of couples equal to 3, 4, 5, ...  the number of seating
   arrangements is
```

> 12, 96, 3120, 115200, 5836320, 382072320, 31488549120, ... (sequence A059375 in the OEIS).

(Wikipedia.)

We can count it using Z3, but also get actual men/women allocations:

```
#!/usr/bin/env python3
from z3 import *

COUPLES=3

# a pair each men and women related to:
men=[Int('men_%d' % i) for i in range(COUPLES)]
women=[Int('women_%d' % i) for i in range(COUPLES)]

# men and women are placed around table like this:

# m m m
#  w w w

# i.e., women[0] is placed between men[0] and men[1]
# the last women[COUPLES-1] is between men[COUPLES-1] and men[0] (wrapping)

s=Solver()
s.add(Distinct(men))
s.add(Distinct(women))

[s.add(And(men[i]>=0, men[i]<COUPLES)) for i in range (COUPLES)]
[s.add(And(women[i]>=0, women[i]<COUPLES)) for i in range (COUPLES)]

# a pair, each woman belong to, cannot be the same as men's located at left and right.
# "% COUPLES" is wrapping, so that the last woman is between the last man and the first man.
for i in range(COUPLES):
    s.add(And(women[i]!=men[i], women[i]!=men[(i+1) % COUPLES]))

def print_model(m):
    print ("  men", end=' ')
    for i in range(COUPLES):
        print (m[men[i]], end=' ')
    print ("")

    print ("women ", end=' ')

    for i in range(COUPLES):
        print (m[women[i]], end=' ')
    print ("")
    print ("")

results=[]

# enumerate all possible solutions:
while True:
    if s.check() == sat:
        m = s.model()
        print_model(m)
        results.append(m)
        block = []
        for d in m:
            c=d()
            block.append(c != m[d])
        s.add(Or(block))
    else:
        print ("results total=", len(results))
        print ("however, according to https://oeis.org/A059375 :", len(results)*2)
        break
```

(https://smt.st/current_tree/other/menage/menage.py)

For 3 couples:

```
  men 0 2 1
women  1 0 2

  men 1 2 0
women  0 1 2

  men 0 1 2
women  2 0 1

  men 2 1 0
women  0 2 1

  men 2 0 1
women  1 2 0

  men 1 0 2
women  2 1 0

results total= 6
however, according to https://oeis.org/A059375 : 12
```

We are getting “half” of results because men and women can be then swapped (their sex swapped (or reassigned)) and you’ve got another 6 results. 6+6=12 in total. This is kind of symmetry.

For 4 couples:

```

...

  men 3 0 2 1
women  1 3 0 2

  men 3 0 1 2
women  2 3 0 1

  men 1 0 2 3
women  3 1 0 2

  men 2 0 1 3
women  3 2 0 1

results total= 48
however, according to https://oeis.org/A059375 : 96
```

For 5 couples:

```

...

  men 0 4 1 2 3
women  1 3 0 4 2

  men 0 3 1 2 4
women  1 4 0 3 2
```

```
  men  0  3  1  2  4
women   1  0  4  3  2

  men  4  3  1  0  2
women   0  2  4  1  3

results total= 1560
however, according to https://oeis.org/A059375 : 3120
```

22.4 Dependency graphs and topological sorting

Topological sorting is an operation many programmers well familiar with: this is what "make" tool do when it find an order of items to process. Items not dependent of anything can be processed first. The most dependent items at the end.

Dependency graph is a graph and topological sorting is such a "contortion" of the a graph, when you can see an order of items.

For example, let's create a sample graph in Wolfram Mathematica:

```
In[]:= g = Graph[{7 -> 1, 7 -> 0, 5 -> 1, 3 -> 0, 3 -> 4, 1 -> 2, 1 -> 6,
   1 -> 4, 0 -> 6}, VertexLabels -> "Name"]
```

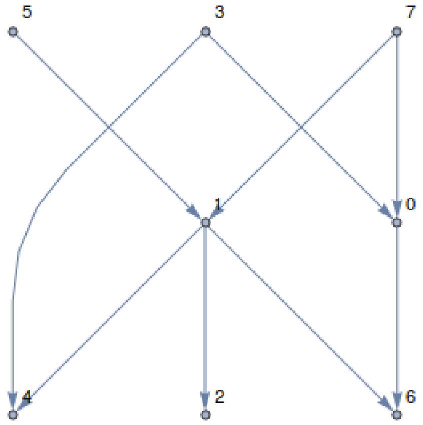


Figure 22.1

Each arrow shows that an item is needed by an item arrow pointing to, i.e., if "a -> b", then item "a" must be first processed, because "b" needs it, or "b" depends on "a".

How Mathematica would "sort" the dependency graph?

```
In[]:= TopologicalSort[g]
Out[]= {7, 3, 0, 5, 1, 4, 6, 2}
```

So you're going to process item 7, then 3, 0, and 2 at the very end.

The algorithm in the Wikipedia article is probably used in the "make" and whatever IDE you use for building your code.

Also, many UNIX platforms had separate "tsort" utility: https://en.wikipedia.org/wiki/Tsort.

How would "tsort" sort the graph? I'm making the text file with input data:

```
7 1
7 0
5 1
3 0
3 4
1 2
1 6
1 4
0 6
```

And run tsort:

```
 % tsort tst
3
5
7
0
1
4
6
2
```

Now I'll use Z3 SMT-solver for topological sort, which is overkill, but quite spectacular: all we need to do is to add constraint for each edge (or "connection") in graph, if "a -> b", then "a" must be less then "b", where each variable reflects ordering.

```
#!/usr/bin/env python3

from z3 import *

TOTAL=8

order=[Int('%d' % i) for i in range(TOTAL)]

s=Solver()
s.add(Distinct(order))
for i in range(TOTAL):
    s.add(And(order[i]>=0, order[i]<TOTAL))

s.add(order[5]<order[1])

s.add(order[3]<order[4])
s.add(order[3]<order[0])

s.add(order[7]<order[0])
s.add(order[7]<order[1])

s.add(order[1]<order[2])
s.add(order[1]<order[4])
s.add(order[1]<order[6])

s.add(order[0]<order[6])

print (s.check())

m=s.model()
order_to_print=[None]*(TOTAL)
for i in range(TOTAL):
    order_to_print[m[order[i]].as_long()]=i

print (order_to_print)
```

Almost the same result, but also correct:

```
True
[5, 7, 1, 3, 0, 4, 6, 2]
```

The solution using MK85: https://smt.st/current_tree/other/tsort/tsort_MK85.py.

Yet another demonstration of topological sort: "less than" relation would indicate, who is whose boss, and who is whose "yes man". The resulting order after sorting is then represent how good each position in social hierarchy is. See also: https://en.wikipedia.org/wiki/Pecking_order.

Another demonstration: when you write a textbook, you first put a material not dependent on anything else. At the end, you put the most advanced material, depending on everything placed before.

Another application: in spreadsheet (3.14), you can reorder all cells in such a way, so that the queue will be started with cells not dependent on anything, etc. And then evaluate cells according to that order.

Another usage as found in Unix v7 man pages:

> The lorder command is essentially obsolete. Use the following command in its place: % ar -ts file.a
>
> The lorder command reads one or more object or library archive files, looks for external references, and writes a list of paired filenames to standard output. The first of each pair of files contains references to identifiers that are defined in the second file. You can send this list to the tsort command to find an ordering of a library member file suitable for 1-pass access by ld.

(https://www.unix.com/man-page/osf1/1/lorder/)

22.5 Package manager

22.5.1 Simple package manager using Z3

Here is simplified example. We have libA, libB, libC and libD, available in various versions (and flavors). We're going to install programA and programB, which use these libraries.

```
#!/usr/bin/env python3

from z3 import *

s=Optimize()

libA=Int('libA')
# libA's version is 1..5 or 999 (which means library will not be installed):
s.add(Or(And(libA>=1, libA<=5),libA==999))

libB=Int('libB')
# libB's version is 1, 4, 5 or 999:
s.add(Or(libB==1, libB==4, libB==5, libB==999))

libC=Int('libC')
# libC's version is 10, 11, 14 or 999:
s.add(Or(libC==10, libC==11, libC==14, libC==999))

# libC is dependent on libA
# libC v10 is dependent on libA v1..3, but not newer
# libC v11 requires at least libA v3
# libC v14 requires at least libA v5
s.add(If(libC==10, And(libA>=1, libA<=3), True))
s.add(If(libC==11, libA>=3, True))
s.add(If(libC==14, libA>=5, True))

libD=Int('libD')
# libD's version is 1..10
s.add(Or(And(libD>=1, libD<=10),libD==999))

programA=Int('programA')
# programA came as v1 or v2:
s.add(Or(programA==1, programA==2))

# programA is dependent on libA, libB and libC
# programA v1 requires libA v2 (only this version), libB v4 or v5, libC v10:
s.add(If(programA==1, And(libA==2, Or(libB==4, libB==5), libC==10), True))
```

```
# programA v2 requires these libraries: libA v3, libB v5, libC v11
s.add(If(programA==2, And(libA==3, libB==5, libC==11), True))

programB=Int('programB')
# programB came as v7 or v8:
s.add(Or(programB==7, programB==8))

# programB v7 requires libA at least v2 and libC at least v10:
s.add(If(programB==7, And(libA>=2, libC>=10), True))
# programB v8 requires libA at least v6 and libC at least v11:
s.add(If(programB==8, And(libA>=6, libC>=11), True))

s.add(programA==1)
s.add(programB==7) # change this to 8 to make it unsat

# we want latest libraries' versions.
# if the library is not required, its version is "pulled up" to 999,
# and 999 means the library is not needed to be installed
s.maximize(Sum(libA,libB,libC,libD))

print (s.check())
print (s.model())
```

(The source code: https://smt.st/current_tree/other/dep/dependency.py)

The output:

```
sat
[libB = 5,
 libD = 999,
 libC = 10,
 programB = 7,
 programA = 1,
 libA = 2]
```

999 means that there is no need to install libD, it's not required by other packages.

Change version of ProgramB to v8 and it will says "unsat", meaning, there is a conflict: ProgramA requires libA v2, but ProgramB v8 eventually requires newer libA.

Still, there is a work to do: "unsat" message is somewhat useless to end user, some information about conflicting items should be printed.

Here is my another optimization problem example: 14.1.

More about using SAT/SMT solvers in package managers: https://research.swtch.com/version-sat, https://cseweb.ucsd.edu/~lerner/papers/opium.pdf.

Now in the opposite direction: forcing aptitude package manager to solve Sudoku:
http://web.archive.org/web/20160326062818/http://algebraicthunk.net/~dburrows/blog/entry/package-managemen

Some readers may ask, how to order libraries/programs/packages to be installed? This is simpler problem, which is often solved by topological sorting. The algorithm reorders graph in such a way so that vertices not depended on anything will be on the top of queue. Next, there will be vertices depended on vertices from the previous layer. And so on.

make UNIX utility does this while constructing order of items to be processed. Even more: older *make* utilities offloaded the job to the external utility *tsort*, which is included in POSIX standard[11].

22.5.2 Toy package manager under 200 SLOC on top of SAT solver

Ubuntu Linux

Many Ubuntu Linux users see this often:

```
% sudo apt install katomic
Reading package lists... Done
Building dependency tree
Reading state information... Done
```

[11] https://pubs.opengroup.org/onlinepubs/9699919799/utilities/tsort.html

```
The following packages were automatically installed and are no longer required:
  a2jmidid libsamplerate0:i386 pulseaudio-module-jack python3-cffi python3-jack-
      client python3-ply python3-pycparser qasmixer
  qastools-common zita-ajbridge
Use 'sudo apt autoremove' to remove them.
The following additional packages will be installed:
  kdoctools5 khelpcenter libgrantlee-templates5 libkf5kdegames-data libkf5kdegames7
      libkf5newstuff-data libkf5newstuff5
  libkf5newstuffcore5 qml-module-org-kde-newstuff
The following NEW packages will be installed:
  katomic kdoctools5 khelpcenter libgrantlee-templates5 libkf5kdegames-data
      libkf5kdegames7 libkf5newstuff-data libkf5newstuff5
  libkf5newstuffcore5 qml-module-org-kde-newstuff
0 upgraded, 10 newly installed, 0 to remove and 20 not upgraded.
Need to get 7,175 kB of archives.
After this operation, 28.3 MB of additional disk space will be used.
Do you want to continue? [Y/n]
```

During installation of a package, other packages are to be installed - libraries, optional packages, etc. Also, 'apt' utility tries to find the most fresh packages.

How an Ubuntu package is defined?

```
Format: 3.0 (quilt)
Source: katomic
Binary: katomic
Architecture: any
Version: 4:20.08.0-1
...
Build-Depends: cmake (>= 3.5~), debhelper-compat (= 13), extra-cmake-modules (>=
    5.30.0~), gettext, libkf5config-dev (>= 5.30.0~), libkf5coreaddons-dev (>= 5.30.0~
    ), libkf5crash-dev (>= 5.30.0~), libkf5dbusaddons-dev (>= 5.30.0~), libkf5doctools
    -dev (>= 5.30.0~), libkf5i18n-dev (>= 5.30.0~), libkf5kdegames-dev (>= 4.9.0~),
    libkf5newstuff-dev (>= 5.30.0~), libkf5widgetsaddons-dev (>= 5.30.0~),
    libkf5xmlgui-dev (>= 5.30.0~), pkg-kde-tools (>> 0.15.15), qtbase5-dev (>= 5.7.0~)
Package-List:
 katomic deb games optional arch=any
...
```

(src)

Here you see that the 'katomic' package requires the 'debhelper-compat' package of exact version 13, 'pkg-kde-tools' newer than version 0.15.15, 'gettext' package of unspecific version, etc...

More on .dsc files file format: 1, 2, 3, 4.

Almost all Linux distributions has some kind of this system. Haskell has 'Cabal' system example package.

This task is a good specimen of CSP, which is NP-problem, of course.

My JSON file format

My toy package manager uses config in JSON format.

This is example of `test100/vers.json` (available versions for each package). (I package numbers in [0..<infinity>) limit and version numbers in [2000...2020] limit.)

```
...
        "8" : [2005, 2006, 2008, 2010, 2011, 2014, 2019],
        "9" : [2000, 2002, 2003, 2004, 2007, 2011, 2015, 2017],
...
```

This is `test100/deps.json` file (dependencies):

```
...
"8":
{
    "2005" : [[1, 0, 2005], [3, 0, 9999], [7, 0, 2005]],
    "2006" : [[1, 0, 9999], [3, 2005, 9999], [7, 0, 2006]],
    "2008" : [[1, 0, 2008], [3, 0, 2008], [7, 0, 9999]],
    "2010" : [[1, 0, 9999], [3, 0, 9999], [7, 2010, 9999]],
    "2011" : [[1, 0, 9999], [3, 0, 2011], [7, 0, 9999]],
    "2014" : [[1, 2008, 9999], [3, 2012, 9999], [7, 2010, 9999]],
    "2019" : [[1, 2019, 9999], [3, 0, 9999], [7, 0, 2019]]
},
"9":
{
    "2000" : [[5, 2000, 9999], [6, 2000, 9999], [7, 0, 9999], [8, 2000, 9999]],
    "2002" : [[5, 0, 9999], [6, 2002, 9999], [7, 0, 9999], [8, 2001, 9999]],
    "2003" : [[5, 2001, 9999], [6, 0, 2003], [8, 2003, 9999]],
    "2004" : [[5, 0, 2004], [6, 2004, 9999], [8, 0, 9999]],
    "2007" : [[5, 2007, 2007], [6, 0, 2007], [8, 0, 9999]],
    "2011" : [[4, 0, 9999], [5, 2002, 9999], [6, 0, 9999], [8, 0, 2011]],
    "2015" : [[4, 0, 9999], [5, 2007, 9999], [6, 0, 9999], [8, 0, 2015]],
    "2017" : [[4, 0, 9999], [5, 2010, 9999], [6, 0, 9999], [8, 0, 9999]]
},
...
```

It defines a list of dependencies for each package/version. Each dependency is a list of triplets. Each triple is: package number, minimal version allowed, maximal version allowed. 0 and 9999 are allowed for min/max versions.

And this a `test100/conflicts.json` file, that lists of conflicting packages:

```
[
[[40, 2013, 2017], [68, 2015, 2020]],
[[76, 2000, 2010], [32, 2000, 2012]],
[[40, 2013, 2017], [71, 2002, 2002]],
[[72, 2004, 2007], [15, 2001, 2016]],
[[36, 2001, 2007], [21, 2000, 2010]],
[[98, 2007, 2010], [14, 2004, 2012]],

...
```

This means, package number 80 in versions [2011..2012] cannot be installed with package 45 in versions [2005..2017]. However, other versions may be allowed.

Why JSON? To make hand editing possible.

In my previous implementation of my toy package manager in Racket, I used S-expressions, but they are less familiar to a general audience. Another option is XML, but it's too verbose...

Version 1: just get a solution

I'm assigning a boolean variable to each package/version.

Making only one version per package possible

First, only one version per package is allowed, so I'm using one-hot encoding and 'AtMost1' constraint for each package. However, a bit may be absent at all. It will leave version value in undefined state.

IMPLY constraint

Second, for each dependency, I'm adding an IMPLY constraint. If a package 1 version 2004 requires package 2 versions 2000..2002, the following constraint is to be added:

```
pkg-1-ver-2004 => pkg-2-ver-2000 OR pkg-2-ver-2001 OR pkg-2-ver-2002
```

($\implies$ is a sign for logical implication.)

That means that if pkg-1-ver-2004=True, one of pkg-2-... must be also true. However, converse is not required. It's possible that pkg-2-ver-2002=True, but pkg-1-ver-2004=False.

See the wikipedia page about it[12].

For a common computer programmer, this can be explained in terms of algol-like programming language:

[12] https://en.wikipedia.org/wiki/Material_conditional

```
if pkg-1-ver-2004:
    set True to one of these: pkg-2-ver-2000 OR pkg-2-ver-2001 OR pkg-2-ver-2002
if another package/version:
    set True to ...
```

... but these things wouldn't work in reverse.

Handling conflicts

For example, package 1 versions [2000..2001] cannot be installed simultaneously with package 2 versions [2005..2007]. I'll add this constraint:

```
AND(pkg-1-ver-2000 OR pkg-1-ver-2001, pkg-2-ver-2005 OR pkg-2-ver-2006 OR pkg-2-ver
    -2007)=False
```

...or...

```
NOT(AND(pkg-1-ver-2000 OR pkg-1-ver-2001, pkg-2-ver-2005 OR pkg-2-ver-2006 OR pkg-2-
    ver-2007))==True
```

NAND gate can also be called "not both". In my example, in plain English language, that means: "pkg-1-ver-2000 OR pkg-1-ver-2001 cannot be true if pkg-2-ver-2005 OR pkg-2-ver-2006 OR pkg-2-ver-2007 is also true and vice versa".

In other words, these two OR expressions cannot be both true, but can be both false.

Getting solution

You'll get a correct solution, but other package/versions may also be turned on. Because you can't 'ground' them to false by default. 'Unconnected packages' are 'dangling' chaotically each time you run SAT solver.

My idea is to collect information from solution recursively, starting at the initial list of desired packages supplied by an end-user. See the `collect_from_solution_recursively()` function.

Test

Let's take a test data from 'test100' folder and run it:

```
% python3 v1.py test100 0 1 5 10 50
get_first_solution
initial_pkgs: [0, 1, 5, 10, 50]
going to run solver
SAT
first solution:
0:2018; 1:2020; 2:2019; 3:2019; 4:2020; 5:2015; 6:2016; 7:2020; 8:2014; 10:2015;
    11:2018; 12:2019; 13:2020; 14:2012; 15:2020; 16:2017; 17:2019; 18:2020; 19:2018;
    21:2020; 22:2000; 23:2001; 28:2001; 29:2020; 34:2020; 35:2014; 36:2015; 50:2014;
```

The solution is correct, but versions are 'random' (in allowed limits), not maximized.

Listing 22.2: The source code

```
#!/usr/bin/env python3

# the code is a bit Lispy, because it was rewritten from Racket...

# no maximize, no MUS, just get first solution

import my_utils, SAT_lib, sys, json
from collections import defaultdict

path=sys.argv[1]

deps=my_utils.string_keys_to_integers(json.load(open(path+"/deps.json")))
vers=my_utils.string_keys_to_integers(json.load(open(path+"/vers.json")))
conflicts=json.load(open(path+"/conflicts.json"))
```

```
packages_total=max(vers.keys())+1

def collect_from_solution_recursively (s, vars, sol, p):
    if p in sol.keys():
        return # already in
    for v in deps[p].keys():
        if (s.get_var_from_solution(vars[p][v])):
            sol[p]=v
            [collect_from_solution_recursively (s, vars, sol, dep[0]) for dep in deps[p][v]]

def get_first_solution(initial_pkgs):
    print ("get_first_solution")
    print ("initial_pkgs:", initial_pkgs)

    s=SAT_lib.SAT_lib(SAT_solver="libpicosat")
    #s=SAT_lib.SAT_lib(SAT_solver="picosat") # slightly faster!

    vars=defaultdict(dict)
    for p in range(packages_total):
        for v in vers[p]:
            vars[p][v]=s.create_var()

    # each variable is one-hot
    [s.AtMost1(list(vars[p].values())) for p in range(packages_total)]

    # ver_lo/ver_high can accept values like 0000 and 9999:
    def version_range_to_list_of_vars(pkg, ver_lo, ver_high):
        rt=[]
        # key=version, val=(str) SAT var
        vers=vars[pkg]
        # find 1st element >= ver_lo
        x=my_utils.find_1st_elem_GE (list(vers.keys()), ver_lo)
        assert x!=None
        # find 1st element <= ver_high
        y=my_utils.find_1st_elem_LE (list(vers.keys()), ver_high)
        assert y!=None
        return list(range(vers[x], vers[y]+1))

    for p in range(0, packages_total):
        for v in deps[p].keys():
            tmp=[s.OR_list(version_range_to_list_of_vars(dep[0], dep[1], dep[2])) for dep in deps[p][v]]
            if len(tmp)>0:
                s.IMPLY_always(vars[p][v], s.AND_list(tmp))

    for pkg in initial_pkgs:
        assert pkg < packages_total
        SAT_vars_to_be_ORed=[vars[pkg][v] for v in deps[pkg].keys()]
        s.fix(s.OR_list(SAT_vars_to_be_ORed), True)

    # add conflicts
    for conflict in conflicts:
        c1, c2 = conflict[0], conflict[1]
        vars1=version_range_to_list_of_vars(c1[0], c1[1], c1[2])
        vars2=version_range_to_list_of_vars(c2[0], c2[1], c2[2])
        s.fix(s.NAND(s.OR_list(vars1), s.OR_list(vars2)), True) # not both

    print ("going to run solver")
    if s.solve():
        print ("SAT")
        sol={}
        [collect_from_solution_recursively (s, vars, sol, p) for p in initial_pkgs]
        return s, vars, sol
    else:
        print ("UNSAT")
        exit(0)

packages_to_install=my_utils.list_of_strings_to_list_of_ints(sys.argv[2:])

# fix packages_to_install. get initial solution
s, vars, solution=get_first_solution (packages_to_install)

print ("first solution:")
print ("; ".join([str(s)+":"+str(solution[s]) for s in sorted(solution.keys())])+"; "
    )
```

Circular dependencies

As a by-product, my toy package manager can support circular dependencies.

Open this example (deps.json) in the test_circ folder:

```
{
"0":
{
    "2016" : [[2,2005,2015]]
},
"1":
{
    "2018" : [[0,2001,2016]]
},
"2":
{
    "2015" : [[1,2006,2018]]
}
}
```

```
% python3 v1.py test_circ 0
get_first_solution
initial_pkgs: [0]
going to run solver
SAT
first solution:
0:2016; 1:2018; 2:2015;
```

Indeed, there are 3 implications: $0 \implies 1$; $1 \implies 2$; $2 \implies 0$. By the rules of boolean logic, that means, all 3 boolean variables would be simultaneously false or true.

I'm not sure if they are handled in a real package manager like Ubuntu's 'apt'. But this feature is possible.

Version 2: maximize versions

The package manager in this form is somewhat useless — an end-user would want to get newest version for each package. So we would maximize them.

This problem seems to be natural to be solved using MaxSAT solver, and I tried, but stuck with a problem: MaxSAT solver tries to raise all package's versions, including 'invisible' ones, not limiting to the ones that has been picked during solving.

This is the algorithm I've conceived:

```
Get an initial solution (like in version 1);

Pick a package/version from solution, fix it at highest version, check
if SAT:
    Leave it as is, pick another package/version
if UNSAT:
    Decrement current version until SAT
```

Example trace:

```
% python3 v2.py test100 5
get_first_solution
initial_pkgs: [5]
going to run solver
SAT
first solution:
0:2018; 5:2006;
trying version 2015 for package 5
SAT
solution: 0:2018; 1:2020; 4:2020; 5:2015;
trying version 2020 for package 4
fixed_versions: {5: 2015}
SAT
solution: 0:2018; 1:2020; 4:2020; 5:2015;
trying version 2020 for package 1
fixed_versions: {5: 2015, 4: 2020}
SAT
solution: 0:2018; 1:2020; 4:2020; 5:2015;
trying version 2018 for package 0
fixed_versions: {5: 2015, 4: 2020, 1: 2020}
SAT
solution: 0:2018; 1:2020; 4:2020; 5:2015;
final solution: 0:2018; 1:2020; 4:2020; 5:2015;
```

Now here I'm trying to 'install' package 4 and 11. My toy package manager tries to raise versions of all packages, including dependent ones:

```
% python3 v2.py test100 4 11
get_first_solution
initial_pkgs: [4, 11]
going to run solver
SAT
first solution:
4:2020; 11:2007;
trying version 2018 for package 11
SAT
solution: 0:2018; 1:2006; 4:2020; 5:2015; 11:2018;
trying version 2015 for package 5
fixed_versions: {11: 2018}
SAT
solution: 0:2018; 1:2006; 4:2020; 5:2015; 11:2018;
trying version 2020 for package 4
fixed_versions: {11: 2018, 5: 2015}
SAT
solution: 0:2018; 1:2006; 4:2020; 5:2015; 11:2018;
trying version 2020 for package 1
fixed_versions: {11: 2018, 5: 2015, 4: 2020}
SAT
solution: 0:2018; 1:2020; 4:2020; 5:2015; 11:2018;
trying version 2018 for package 0
fixed_versions: {11: 2018, 5: 2015, 4: 2020, 1: 2020}
SAT
solution: 0:2018; 1:2020; 4:2020; 5:2015; 11:2018;
final solution: 0:2018; 1:2020; 4:2020; 5:2015; 11:2018;
```

By the way, it's important to note that the first solution can coincide with the 'best' solution. But may not.

Listing 22.3: The source code

```
#!/usr/bin/env python3

# the code is a bit Lispy, because it was rewritten from Racket...

# maximize package versions, no MUS

import my_utils, SAT_lib, sys, json
from collections import defaultdict

path=sys.argv[1]

deps=my_utils.string_keys_to_integers(json.load(open(path+"/deps.json")))
vers=my_utils.string_keys_to_integers(json.load(open(path+"/vers.json")))
conflicts=json.load(open(path+"/conflicts.json"))

packages_total=max(vers.keys())+1

def collect_from_solution_recursively (s, vars, sol, p):
    if p in sol.keys():
        return # already in
    for v in deps[p].keys():
        if (s.get_var_from_solution(vars[p][v])):
            sol[p]=v
            [collect_from_solution_recursively (s, vars, sol, dep[0]) for dep in deps
               [p][v]]

def get_first_solution(initial_pkgs):
    print ("get_first_solution")
    print ("initial_pkgs:", initial_pkgs)

    s=SAT_lib.SAT_lib(SAT_solver="libpicosat")

    vars=defaultdict(dict)
    for p in range(packages_total):
        for v in vers[p]:
            vars[p][v]=s.create_var()

    # each variable is one-hot
    [s.AtMost1(list(vars[p].values())) for p in range(packages_total)]

    # ver_lo/ver_high can accept values like 0000 and 9999:
    def version_range_to_list_of_vars(pkg, ver_lo, ver_high):
        rt=[]
        # key=version, val=(str) SAT var
        vers=vars[pkg]
        # find 1st element >= ver_lo
        x=my_utils.find_1st_elem_GE (list(vers.keys()), ver_lo)
        assert x!=None
        # find 1st element <= ver_high
        y=my_utils.find_1st_elem_LE (list(vers.keys()), ver_high)
        assert y!=None
        return list(range(vers[x], vers[y]+1))

    for p in range(0, packages_total):
        for v in deps[p].keys():
            tmp=[s.OR_list(version_range_to_list_of_vars(dep[0], dep[1], dep[2])) for
                 dep in deps[p][v]]
            if len(tmp)>0:
                s.IMPLY_always(vars[p][v], s.AND_list(tmp))
    for pkg in initial_pkgs:
        assert pkg < packages_total
        SAT_vars_to_be_ORed=[vars[pkg][v] for v in deps[pkg].keys()]
        s.fix(s.OR_list(SAT_vars_to_be_ORed), True)

    # add conflicts
    for conflict in conflicts:
        c1, c2 = conflict[0], conflict[1]
        vars1=version_range_to_list_of_vars(c1[0], c1[1], c1[2])
        vars2=version_range_to_list_of_vars(c2[0], c2[1], c2[2])
        s.fix(s.NAND(s.OR_list(vars1), s.OR_list(vars2)), True) # not both

    print ("going to run solver")
    if s.solve():
        print ("SAT")
        sol={}
        [collect_from_solution_recursively (s, vars, sol, p) for p in initial_pkgs]
        return s, vars, sol
    else:
        print ("UNSAT")
        exit(0)

packages_to_install=my_utils.list_of_strings_to_list_of_ints(sys.argv[2:])

# fix packages_to_install. get initial solution
s, vars, solution=get_first_solution (packages_to_install)

print ("first solution:")
print ("; ".join([str(s)+":"+str(solution[s]) for s in sorted(solution.keys())])+"; "
   )

fixed_versions={}

def find_max_version_for_package(solution, s, vars, pkg, fixed_versions):
    found=False
    for v in vers[pkg][::-1]: # from highest to lowest
        print ("trying version", v, "for package", pkg)
        if len(fixed_versions)>0:
            print ("fixed_versions:", fixed_versions)
        SAT_var=vars[pkg][v]
        s.assume(SAT_var) # assume it's true

        res=s.solve()

        if res==False:
            print ("UNSAT")
            continue
        else:
            # new solution
            solution={}
            [collect_from_solution_recursively (s, vars, solution, p) for p in
                packages_to_install]
            print ("SAT")
            t="; ".join([str(s)+":"+str(solution[s]) for s in sorted(solution.keys())
                ])+"; "
            print ("solution: "+t)
            # we stop here
            found=True
            fixed_versions[pkg]=v
            s.fix(SAT_var, True) # fix this package/version until the end
            return solution
    assert found

while True:
    # find a package that present in solution, but not in fixed_versions
    # (by computing set difference)
    in_solution_but_not_in_fixed_versions=set(solution.keys()) - set(fixed_versions.
        keys())
    # if we can't find one, finish
    if len(in_solution_but_not_in_fixed_versions)==0:
        s="; ".join([str(s)+":"+str(solution[s]) for s in sorted(solution.keys())])+"
            ; "
        print ("final solution: "+s)
        exit(0)
    else:
        # found
        # pick a package with highest number
        # (we start enumerating all versions at the highest package number)
        pkg=max(in_solution_but_not_in_fixed_versions)
        solution=find_max_version_for_package(solution, s, vars, pkg, fixed_versions)
```

SAT assumptions

I'm using SAT assumptions. What is this?

Imagine you play Doom/Quake/another shooter videogame. You gone far and you stumbled before the fork in the road. You know you may get killed by a monster. So you save the state of the game, you take a first path, you get killed by a monster, you restore a state, you take a second path, etc, etc. You want to save the state because you don't want to lose what you've achieved up to this point.

It takes a time for SAT solver to process input clauses, to 'warm up', like an engine.

Here I'm using SAT assumption after all the constraints processed, and I 'assume' that a package X has version Y. I execute sat() call of picosat, it returns SAT or UNSAT, and all 'assumptions' are reset after each sat() call, but the picosat SAT solver is still 'warmed up', and all constraints are still there. And then I 'assume' that a package X has version Y-1, and execute sat() call again, etc, etc.

It's impossible to add assumptions running picosat executable, so I have to load the libpicosat.so shared library and call all these functions one after another.

It's important to note that I could run picosat executable instead, without any assumptions, but then the whole process would be longer: because you'll have to generate CNF file each time, picosat SAT solver have to process all the clauses from the beginning, 'warm its engine', etc.

Version 3: MUS: get info about conflicts

In case of conflict, my toy package manager just gives "UNSAT", which is probably not a very useful response. Can we get more information, which packages conflicts?

I use PicoMUS utility here.

The PicoMUS utility from picosat SAT solver suite will tell, which CNF clauses are to be removed to make UNSAT instance SAT. But for that we need to track, which clauses may be related to which packages.

For example, when IMPLY constraints are added, they clause numbers (from `c.CNF_next_idx`) are saved into a dictionary:

```
    for p in range(0, packages_total):
        for v in deps[p].keys():
            tmp=[s.OR_list(version_range_to_list_of_vars(dep[0], dep[1], dep[2])) for
                 dep in deps[p][v]]
            if len(tmp)>0:
                if get_MUS:
                    clause_start=s.CNF_next_idx+1
                s.IMPLY_always(vars[p][v], s.AND_list(tmp))
                if get_MUS:
                    clause_stop=s.CNF_next_idx-1+1
                    #print ("IMPLY", p, v, "clauses=[", clause_start, clause_stop,
                        "]")
                        [IMPLY_clauses_packages[c].add(p) for c in range(clause_start,
                            clause_stop+1)]
```

And when PicoMUS utility returns list of clauses, we get a list of packages from our records, that are connected to these clauses:

```
        print ("running picomus")
        MUS_clauses, MUS_vars=s.get_MUS_vars()
...
        IMPLY_clauses_packages_out=set()
...
        for c in MUS_clauses:
...
            if c in IMPLY_clauses_packages.keys():
                IMPLY_clauses_packages_out.update(IMPLY_clauses_packages[c])
...
        print ("conflicted packages (IMPLY):", sorted(list(IMPLY_clauses_packages_out
            )))
```

We also keep tabs on clauses added when fixing packages user wants to install and clauses generated during conflict.json file processing.

Now a small example.

Listing 22.4: vers.json

```
{
        "0" : [2011, 2016],
        "1" : [2013, 2018],
        "2" : [2010, 2015]
}
```

Listing 22.5: deps.json

```
{
"0":
{
    "2011" : [],
    "2016" : []
},
"1":
{
    "2013" : [],
    "2018" : []
},
"2":
{
    "2010" : [],
    "2015" : []
}
}
```

Listing 22.6: conflicts.json

```
[
[[1, 2013, 2018], [2, 2010, 2015]]
]
```

And I'm trying to 'install' packages 0, 1 and 2, which is impossible:

```
% python3 v3.py test2 0 1 2
get_first_solution, get_MUS= False
initial_pkgs: [0, 1, 2]
going to run solver
UNSAT
```

```
get_first_solution, get_MUS= True
initial_pkgs: [0, 1, 2]
running picomus
conflicted packages (set by user): [1, 2]
conflicted packages (IMPLY): []
conflicted packages (from conflicts.json): [1, 2]
```

Now you see that packages 1 and 2 cannot be installed with each other.

Another example (test6). Package 1 requires package 0 of version 2014 (only this version) and package 2 requires package 0 version 2011 only:

```
{
"0":
{
    "2011" : [],
    "2014" : []
},
"1":
{
    "2018" : [[0,2014,2014]]
},
"2":
{
    "2015" : [[0,2011,2011]]
}
}
```

My toy package manager find the conflict:

```
% python3 v3.py test6 1 2
get_first_solution, get_MUS= False
initial_pkgs: [1, 2]
going to run solver
UNSAT
get_first_solution, get_MUS= True
initial_pkgs: [1, 2]
running picomus
conflicted packages (set by user): [1, 2]
conflicted packages (IMPLY): [1, 2]
conflicted packages (from conflicts.json): []
```

Now example from my randomly generated test data (test100):

```
% python3 v3.py test200 138
get_first_solution, get_MUS= False
initial_pkgs: [138]
going to run solver
UNSAT
get_first_solution, get_MUS= True
initial_pkgs: [138]
running picomus
conflicted packages (set by user): [138]
conflicted packages (IMPLY): [1, 2, 3, 4, 5, 6, 8, 9, 11, 12, 13, 14, 15, 16, 21, 22,
    28, 30, 31, 32, 34, 35, 37, 40, 44, 45, 50, 51, 54, 56, 58, 59, 61, 63, 70, 78,
   81, 83, 85, 97, 98, 126, 128, 137, 138]
conflicted packages (from conflicts.json): [12, 30, 52, 61]
```

Unfortunately, PicoMUS is slow and usable only for debugging. It may take minutes to find a MUS.

On my Intel Core 2 Duo CPU clocked at 2.13GHz:

```
python3 v3.py test200 187  102.24s user 0.15s system 99% cpu 1:43.05 total
python3 v3.py test500 492  101.44s user 0.17s system 99% cpu 1:41.92 total
python3 v3.py test1000 980  38.11s user 0.17s system 99% cpu 38.602 total
```

It's unlikely, an end user would wait this long. But my tests are much more complex than real-world cases, such as Ubuntu packages, etc. So it can be usable.

Listing 22.7: The source code

```
#!/usr/bin/env python3

# the code is a bit Lispy, because it was rewritten from Racket...

# full version: maximize package versions, use MUS if conflict

import my_utils, SAT_lib, sys, json
from collections import defaultdict

path=sys.argv[1]

deps=my_utils.string_keys_to_integers(json.load(open(path+"/deps.json")))
vers=my_utils.string_keys_to_integers(json.load(open(path+"/vers.json")))
conflicts=json.load(open(path+"/conflicts.json"))

packages_total=max(vers.keys())+1

def collect_from_solution_recursively (s, vars, sol, p):
    if p in sol.keys():
        return # already in
    for v in deps[p].keys():
        if (s.get_var_from_solution(vars[p][v])):
            sol[p]=v
            [collect_from_solution_recursively (s, vars, sol, dep[0]) for dep in deps
               [p][v]]

def get_first_solution(initial_pkgs, get_MUS):
    print ("get_first_solution, get_MUS=", get_MUS)
    print ("initial_pkgs:", initial_pkgs)

    if get_MUS:
        s=SAT_lib.SAT_lib(SAT_solver="picomus")
    else:
        s=SAT_lib.SAT_lib(SAT_solver="libpicosat")

    vars=defaultdict(dict)
    for p in range(packages_total):
        for v in vers[p]:
            vars[p][v]=s.create_var()

    if get_MUS:
        fix_clauses_packages=defaultdict(set)
        IMPLY_clauses_packages=defaultdict(set)
        conflicts_clauses_packages=defaultdict(set)

    # each variable is one-hot
    [s.AtMost1(list(vars[p].values())) for p in range(packages_total)]

    # ver_lo/ver_high can accept values like 0000 and 9999:
    def version_range_to_list_of_vars(pkg, ver_lo, ver_high):
        rt=[]
        # key=version, val=(str) SAT var
        vers=vars[pkg]
        # find 1st element >= ver_lo
        x=my_utils.find_1st_elem_GE (list(vers.keys()), ver_lo)
        assert x!=None
        # find 1st element <= ver_high
        y=my_utils.find_1st_elem_LE (list(vers.keys()), ver_high)
        assert y!=None
        return list(range(vers[x], vers[y]+1))

    for p in range(0, packages_total):
        for v in deps[p].keys():
            tmp=[s.OR_list(version_range_to_list_of_vars(dep[0], dep[1], dep[2])) for
                dep in deps[p][v]]
            if len(tmp)>0:
                if get_MUS:
                    clause_start=s.CNF_next_idx+1
                s.IMPLY_always(vars[p][v], s.AND_list(tmp))
                if get_MUS:
                    clause_stop=s.CNF_next_idx-1+1
                    #print ("IMPLY", p, v, "clauses=[", clause_start, clause_stop,
                        "]")
                    [IMPLY_clauses_packages[c].add(p) for c in range(clause_start,
                        clause_stop+1)]

    for pkg in initial_pkgs:
        assert pkg < packages_total
        if get_MUS:
            clause_start=s.CNF_next_idx+1
        SAT_vars_to_be_ORed=[vars[pkg][v] for v in deps[pkg].keys()]
        s.fix(s.OR_list(SAT_vars_to_be_ORed), True)
        if get_MUS:
            clause_stop=s.CNF_next_idx-1+1
            #print ("s.fix", pkg, "clauses=[", clause_start, clause_stop, "]")
            [fix_clauses_packages[c].add(pkg) for c in range(clause_start,
                clause_stop+1)]

    # add conflicts
    for conflict in conflicts:
        if get_MUS:
            clause_start=s.CNF_next_idx+1
        c1, c2 = conflict[0], conflict[1]
        vars1=version_range_to_list_of_vars(c1[0], c1[1], c1[2])
        vars2=version_range_to_list_of_vars(c2[0], c2[1], c2[2])
        s.fix(s.NAND(s.OR_list(vars1), s.OR_list(vars2)), True) # not both
        if get_MUS:
            clause_stop=s.CNF_next_idx-1+1
            #print ("conflict between", c1, c2, "clauses=[", clause_start,
                clause_stop, "]")
            for c in range(clause_start, clause_stop+1):
                conflicts_clauses_packages[c].add(c1[0])
                conflicts_clauses_packages[c].add(c2[0])

    if get_MUS:
        print ("running picomus")
        MUS_clauses, MUS_vars=s.get_MUS_vars()
        fix_clauses_packages_out=set()
        IMPLY_clauses_packages_out=set()
        conflicts_clauses_packages_out=set()
        for c in MUS_clauses:
            if c in fix_clauses_packages.keys():
                fix_clauses_packages_out.update(fix_clauses_packages[c])
            if c in IMPLY_clauses_packages.keys():
                IMPLY_clauses_packages_out.update(IMPLY_clauses_packages[c])
            if c in conflicts_clauses_packages.keys():
                conflicts_clauses_packages_out.update(conflicts_clauses_packages[c])
        print ("conflicted packages (set by user):", sorted(list(
           fix_clauses_packages_out)))
        print ("conflicted packages (IMPLY):", sorted(list(IMPLY_clauses_packages_out
           )))
        print ("conflicted packages (from conflicts.json):", sorted(list(
           conflicts_clauses_packages_out)))
        exit(0)

    print ("going to run solver")
    if s.solve():
        print ("SAT")
        sol={}
        [collect_from_solution_recursively (s, vars, sol, p) for p in initial_pkgs]
        return s, vars, sol
    else:
        print ("UNSAT")
        return None, None, None

packages_to_install=my_utils.list_of_strings_to_list_of_ints(sys.argv[2:])

# fix packages_to_install. get initial solution
s, vars, solution=get_first_solution (packages_to_install, get_MUS=False)
if solution==None:
    get_first_solution (packages_to_install, get_MUS=True)
    exit(0)

print ("first solution:")
print ("; ".join([str(s)+":"+str(solution[s]) for s in sorted(solution.keys())])+"; "
   )

fixed_versions={}

def find_max_version_for_package(solution, s, vars, pkg, fixed_versions):
    found=False
    for v in vers[pkg][::-1]: # from highest to lowest
        print ("trying version", v, "for package", pkg)
        if len(fixed_versions)>0:
            print ("fixed_versions:", fixed_versions)
        SAT_var=vars[pkg][v]
        s.assume(SAT_var) # assume it's true

        res=s.solve()

        if res==False:
            print ("UNSAT")
            continue
        else:
            # new solution
            solution={}
            [collect_from_solution_recursively (s, vars, solution, p) for p in
               packages_to_install]
            print ("SAT")
            t="; ".join([str(s)+":"+str(solution[s]) for s in sorted(solution.keys())
               ])+"; "
            print ("solution: "+t)
            # we stop here
            found=True
            fixed_versions[pkg]=v
            s.fix(SAT_var, True) # fix this package/version until the end
            return solution
    assert found

while True:
    # find a package that present in solution, but not in fixed_versions
    # (by computing set difference)
    in_solution_but_not_in_fixed_versions=set(solution.keys()) - set(fixed_versions.keys())
    # if we can't find one, finish
    if len(in_solution_but_not_in_fixed_versions)==0:
        s="; ".join([str(s)+":"+str(solution[s]) for s in sorted(solution.keys())])+"; "
        print ("final solution: "+s)
        exit(0)
    else:
        # found
        # pick a package with highest number
        # (we start enumerating all versions at the highest package number)
        pkg=max(in_solution_but_not_in_fixed_versions)
        solution=find_max_version_for_package(solution, s, vars, pkg, fixed_versions)
```

All the files

https://smt.st/current_tree/other/dep/picosat/files

Future work

'Remove' operation.

Further reading

Russ Cox: 1, 2.

Several attempts exists on adding SAT solvers to package managers in major *NIX distributions:
Fedora-specific: 1, 2.
Debian-specific: 1, 2.
SUSE-specific: 1, 2, 3, 4, 5.
FreeBSD: 1. Has picosat SAT solver: 1.
More comments: Fedora, OpenSUSE, Windows 7, PubGrub and Dart, PHP, Anaconda.
A paper: OPIUM: Optimal Package Install/Uninstall Manager.

22.6 Stable marriage problem

See also in Wikipedia and Rosetta code.

Layman's explanation in Russian: https://lenta.ru/articles/2012/10/15/nobel/.

One interesting use of it:

> The Internet infrastructure company Akamai, cofounded by Tom Leighton, also uses a variation of the Mating Ritual to assign web traffic to its servers.
>
> In the early days, Akamai used other combinatorial optimization algorithms that got to be too slow as the number of servers (over 65,000 in 2010) and requests (over 800 billion per day) increased. Akamai switched to a Ritual-like approach, since a Ritual is fast and can be run in a distributed manner. In this case, web requests correspond to women and web servers correspond to men. The web requests have preferences based on latency and packet loss, and the web servers have preferences based on cost of bandwidth and co-location.

(Eric Lehman, F Thomson Leighton, Albert R Meyer - Mathematics for Computer Science)

My solution is much less efficient, because much simpler/better algorithm exists (Gale/Shapley algorithm), but I did it to demonstrate the essence of the problem plus as a yet another SMT-solvers and Z3 demonstration.

See comments:

```
#!/usr/bin/env python3

from z3 import *

SIZE=10

# names and preferences has been copypasted from https://rosettacode.org/wiki/
   Stable_marriage_problem

# males:
abe, bob, col, dan, ed, fred, gav, hal, ian, jon = 0,1,2,3,4,5,6,7,8,9
MenStr=["abe", "bob", "col", "dan", "ed", "fred", "gav", "hal", "ian", "jon"]
# females:
abi, bea, cath, dee, eve, fay, gay, hope, ivy, jan = 0,1,2,3,4,5,6,7,8,9
WomenStr=["abi", "bea", "cath", "dee", "eve", "fay", "gay", "hope", "ivy", "jan"]

# men's preferences. better is at left (at first):
ManPrefer={}
ManPrefer[abe]=[abi, eve, cath, ivy, jan, dee, fay, bea, hope, gay]
ManPrefer[bob]=[cath, hope, abi, dee, eve, fay, bea, jan, ivy, gay]
ManPrefer[col]=[hope, eve, abi, dee, bea, fay, ivy, gay, cath, jan]
ManPrefer[dan]=[ivy, fay, dee, gay, hope, eve, jan, bea, cath, abi]
ManPrefer[ed]=[jan, dee, bea, cath, fay, eve, abi, ivy, hope, gay]
ManPrefer[fred]=[bea, abi, dee, gay, eve, ivy, cath, jan, hope, fay]
ManPrefer[gav]=[gay, eve, ivy, bea, cath, abi, dee, hope, jan, fay]
ManPrefer[hal]=[abi, eve, hope, fay, ivy, cath, jan, bea, gay, dee]
ManPrefer[ian]=[hope, cath, dee, gay, bea, abi, fay, ivy, jan, eve]
ManPrefer[jon]=[abi, fay, jan, gay, eve, bea, dee, cath, ivy, hope]

# women's preferences:
WomanPrefer={}
WomanPrefer[abi]=[bob, fred, jon, gav, ian, abe, dan, ed, col, hal]
WomanPrefer[bea]=[bob, abe, col, fred, gav, dan, ian, ed, jon, hal]
WomanPrefer[cath]=[fred, bob, ed, gav, hal, col, ian, abe, dan, jon]
WomanPrefer[dee]=[fred, jon, col, abe, ian, hal, gav, dan, bob, ed]
WomanPrefer[eve]=[jon, hal, fred, dan, abe, gav, col, ed, ian, bob]
WomanPrefer[fay]=[bob, abe, ed, ian, jon, dan, fred, gav, col, hal]
WomanPrefer[gay]=[jon, gav, hal, fred, bob, abe, col, ed, dan, ian]
WomanPrefer[hope]=[gav, jon, bob, abe, ian, dan, hal, ed, col, fred]
WomanPrefer[ivy]=[ian, col, hal, gav, fred, bob, abe, ed, jon, dan]
WomanPrefer[jan]=[ed, hal, gav, abe, bob, jon, col, ian, fred, dan]

s=Solver()

ManChoice=[Int('ManChoice_%d' % i) for  i in range(SIZE)]
WomanChoice=[Int('WomanChoice_%d' % i) for  i in range(SIZE)]

# all values in ManChoice[]/WomanChoice[] are in 0..9 range:
for i in range(SIZE):
    s.add(And(ManChoice[i]>=0, ManChoice[i]<=9))
    s.add(And(WomanChoice[i]>=0, WomanChoice[i]<=9))

s.add(Distinct(ManChoice))

# "inverted index", make sure all men and women are "connected" to each other, i.e.,
   form pairs.
# FIXME: only work for SIZE=10
for i in range(SIZE):
    s.add(WomanChoice[i]==
        If(ManChoice[0]==i, 0,
        If(ManChoice[1]==i, 1,
        If(ManChoice[2]==i, 2,
        If(ManChoice[3]==i, 3,
        If(ManChoice[4]==i, 4,
        If(ManChoice[5]==i, 5,
        If(ManChoice[6]==i, 6,
        If(ManChoice[7]==i, 7,
        If(ManChoice[8]==i, 8,
        If(ManChoice[9]==i, 9, -1))))))))))

# this is like ManChoice[] value, but "inverted index". it reflects wife's rating in
   man's own rating system.
# 0 if he married best women, 1 if there is 1 women who he would prefer (if there is
   a chance):
ManChoiceInOwnRating=[Int('ManChoiceInOwnRating_%d' % i) for  i in range(SIZE)]
# same for all women:
WomanChoiceInOwnRating=[Int('WomanChoiceInOwnRating_%d' % i) for  i in range(SIZE)]

# set values in "inverted" indices according to values in ManPrefer[]/WomenPrefer[].
# FIXME: only work for SIZE=10
for m in range(SIZE):
    s.add (ManChoiceInOwnRating[m]==
       If(ManChoice[m]==ManPrefer[m][0],0,
       If(ManChoice[m]==ManPrefer[m][1],1,
       If(ManChoice[m]==ManPrefer[m][2],2,
       If(ManChoice[m]==ManPrefer[m][3],3,
       If(ManChoice[m]==ManPrefer[m][4],4,
       If(ManChoice[m]==ManPrefer[m][5],5,
       If(ManChoice[m]==ManPrefer[m][6],6,
       If(ManChoice[m]==ManPrefer[m][7],7,
       If(ManChoice[m]==ManPrefer[m][8],8,
       If(ManChoice[m]==ManPrefer[m][9],9, -1)))))))))))

for w in range(SIZE):
    s.add (WomanChoiceInOwnRating[w]==
       If(WomanChoice[w]==WomanPrefer[w][0],0,
       If(WomanChoice[w]==WomanPrefer[w][1],1,
       If(WomanChoice[w]==WomanPrefer[w][2],2,
       If(WomanChoice[w]==WomanPrefer[w][3],3,
       If(WomanChoice[w]==WomanPrefer[w][4],4,
       If(WomanChoice[w]==WomanPrefer[w][5],5,
       If(WomanChoice[w]==WomanPrefer[w][6],6,
       If(WomanChoice[w]==WomanPrefer[w][7],7,
       If(WomanChoice[w]==WomanPrefer[w][8],8,
       If(WomanChoice[w]==WomanPrefer[w][9],9, -1)))))))))))

# the last part is the essence of this script:

# this is 2D bool array. "true" if a (married or already connected) man would prefer
   another women over his wife.
ManWouldPrefer=[[Bool('ManWouldPrefer_%d_%d' % (m, w)) for w in range(SIZE)] for m in
    range(SIZE)]
# same for all women:
WomanWouldPrefer=[[Bool('WomanWouldPrefer_%d_%d' % (w, m)) for m in range(SIZE)] for
   w in range(SIZE)]

# set "true" in ManWouldPrefer[][] table for all women who are better than the wife a
    man currently has.
# all others can be "false"
# if the man married best women, all entries would be "false"
for m in range(SIZE):
    for w in range(SIZE):
        s.add(ManWouldPrefer[m][w] == (ManPrefer[m].index(w) < ManChoiceInOwnRating[m
            ]))

# do the same for WomanWouldPrefer[][]:
for w in range(SIZE):
    for m in range(SIZE):
        s.add(WomanWouldPrefer[w][m] == (WomanPrefer[w].index(m) <
            WomanChoiceInOwnRating[w]))

# this is the most important constraint.
# enumerate all possible man/woman pairs
# no pair can exist with both "true" in "mirrored" entries of ManWouldPrefer[][]/
    WomanWouldPrefer[][].
# we block this by the following constraint: Not(And(x,y)): all x/y values are
    allowed, except if both are set to 1/true:
for m in range(SIZE):
    for w in range(SIZE):
        s.add(Not(And(ManWouldPrefer[m][w], WomanWouldPrefer[w][m])))

print (s.check())
mdl=s.model()

print ("")

print ("ManChoice:")
for m in range(SIZE):
    w=mdl[ManChoice[m]].as_long()
    print (MenStr[m], "<->", WomenStr[w])

print ("")

print ("WomanChoice:")
for w in range(SIZE):
    m=mdl[WomanChoice[w]].as_long()
    print (WomenStr[w], "<->", MenStr[m])
```

(The source code: https://smt.st/current_tree/other/stable_marriage/stable.py)

Result is seems to be correct:

```
sat

ManChoice:
abe <-> ivy
bob <-> cath
col <-> dee
dan <-> fay
ed <-> jan
fred <-> bea
gav <-> gay
hal <-> eve
ian <-> hope
jon <-> abi

WomanChoice:
abi <-> jon
bea <-> fred
cath <-> bob
dee <-> col
eve <-> hal
fay <-> dan
gay <-> gav
```

```
hope <-> ian
ivy <-> abe
jan <-> ed
```

This is what we did in plain English language. “Connect men and women somehow, we don’t care how. But no pair must exist of those who prefer each other (simultaneously) over their current spouses”. Gale/Shapley algorithm uses “steps” to “stabilize” marriage. There are no “steps”, all pairs are married couples already.

Another important thing to notice: only one solution must exist.

```
...

results=[]

# enumerate all possible solutions:
while True:
    if s.check() == sat:
        m = s.model()
        #print m
        results.append(m)
        block = []
        for d in m:
            c=d()
            block.append(c != m[d])
        s.add(Or(block))
    else:
        print "results total=", len(results)
        break

...
```

(The source code: https://smt.st/current_tree/other/stable_marriage/stable2.py)

That reports only 1 model available, which is correct indeed.

Wojciech Niedbala fixed chained If’s, but I left the old version, so readers can see a difference...

```
...

def _if_x(x, ind, i):
    if i == len(x) - 1:
        return If(x[i] == ind, i, -1)
    return If(x[i] == ind, i, _if_x(x, ind, i+1))


def _if_xy(x, y, i):
    if i == len(y) - 1:
        return If(x == y[i], i, -1)
    return If(x == y[i], i, _if_xy(x, y, i+1))

...

    s.add(WomanChoice[i]== _if_x(ManChoice, i, 0))

...

    s.add (ManChoiceInOwnRating[m]== _if_xy(ManChoice[m], ManPrefer[m], 0))
```

(The source code: https://smt.st/current_tree/other/stable_marriage/stable_fixed.py)

22.7 Tiling puzzle

22.7.1 Tiling puzzle (SMT)

This is classic problem: given 12 polyomino titles, cover mutilated chessboard with them (it has 60 squares with no central 4 squares).

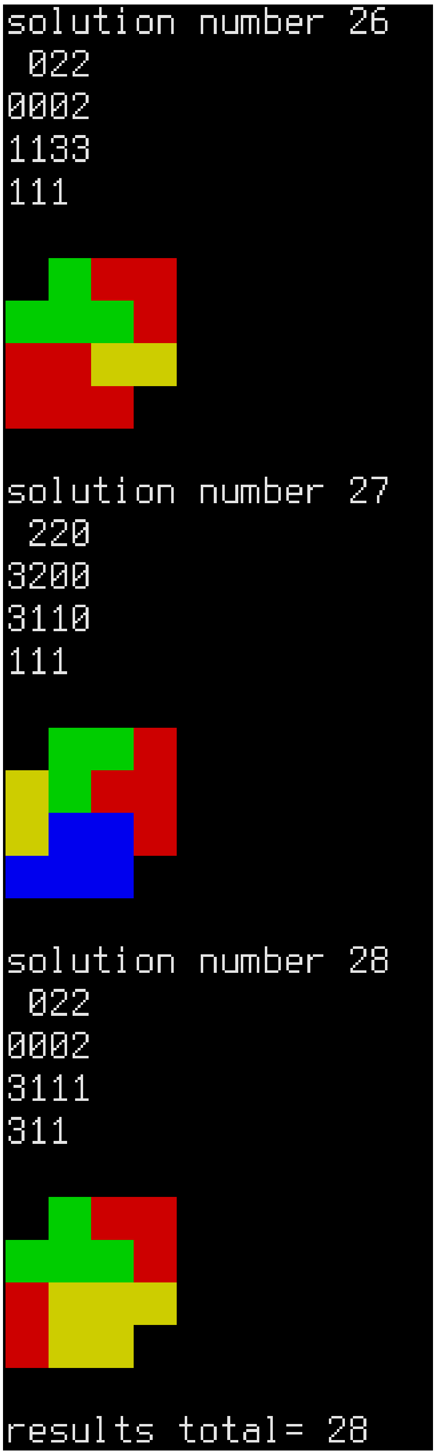
The problem is covered at least in Donald E. Knuth - Dancing Links, and this Z3 solution has been inspired by it.

Another thing I've added: graph coloring (9). You see, my script gives correct solutions, but somewhat unpleasant visually. So I used colored pseudographics. There are 12 tiles, it's not a problem to assign 12 colors to them. But there is another popular SAT problem: graph coloring.

Given a graph, assign a color to each vertex/node, so that colors wouldn't be equal in adjacent nodes. The problem can be solved easily in SMT: assign variable to each vertex. If two vertices are connected, add a constraint: *vertex1_color != vertex2_color*. As simple as that. In my case, each polyomino is vertex and if polyomino is adjacent to another polyomino, an edge/link is added between vertices. So I did, and output is now colored.

But this is planar graph (i.e., a graph which is, if represented in two-dimensional space has no intersected edges/links). And here is a famous four color theorem can be used. The solution of tiled polyominos is in fact like planar graph, or, a map, like a world map. Theorem states that any planar graph (or map) can be colored only 4 colors.

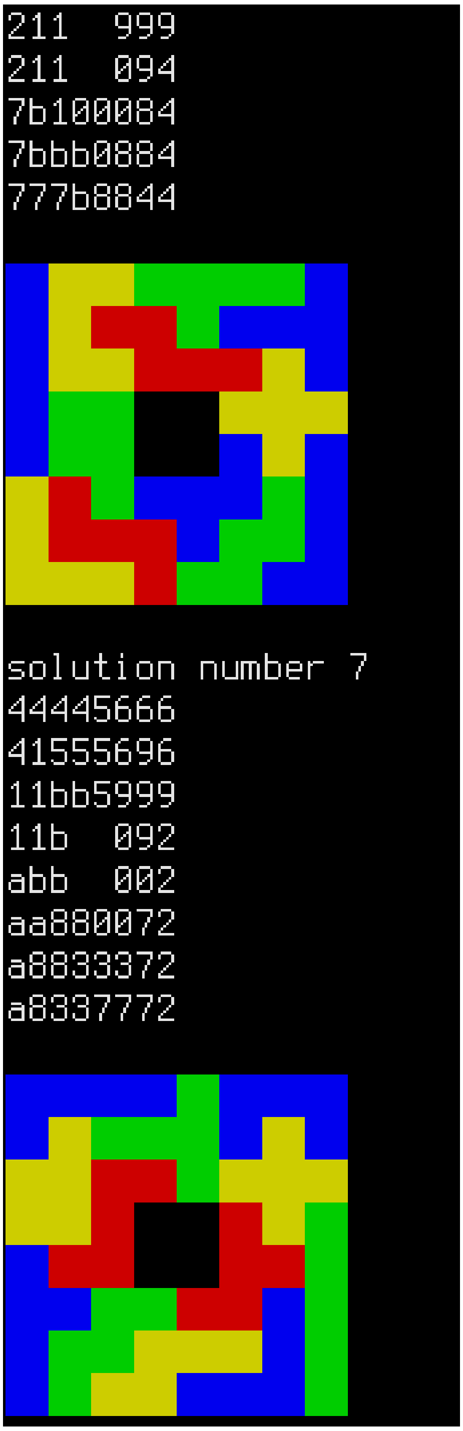
This is true, even more, several tilings can be colors with only 3 colors:



```
solution number 26
 022
0002
1133
111


solution number 27
 220
3200
3110
111


solution number 28
 022
0002
3111
311


results total= 28
```

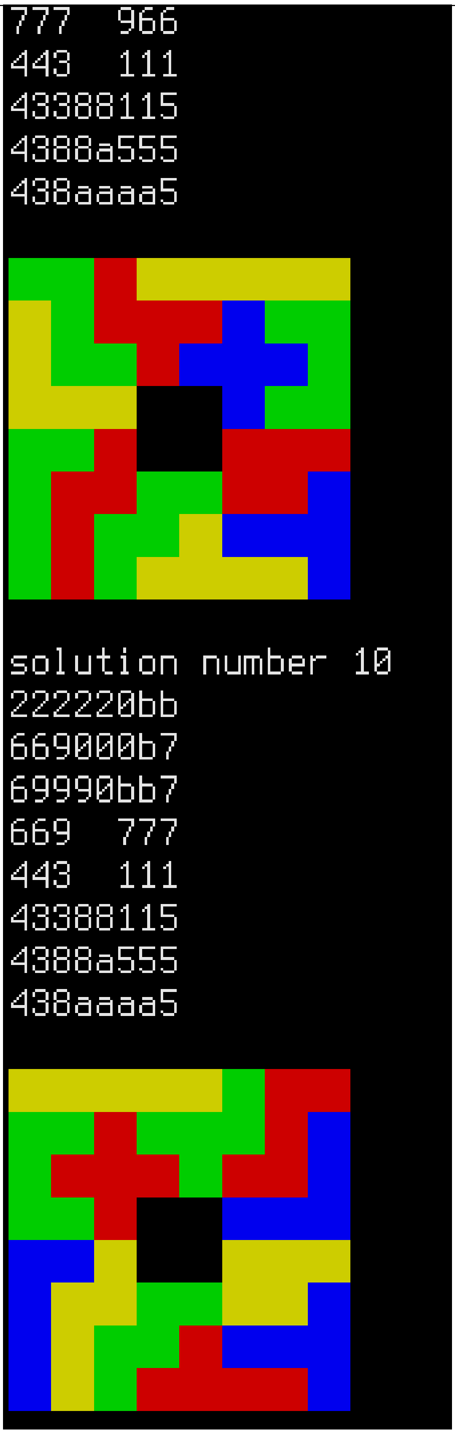
Figure 22.2

Now the classic: 12 pentominos and ”mutilated” chess board, several solutions:



```
211  999
211  094
7b100084
7bbb0884
777b8844

solution number 7
44445666
41555696
11bb5999
11b  092
abb  002
aa880072
a8833372
a8337772
```

Figure 22.3



```
777  966
443  111
43388115
4388a555
438aaaa5

solution number 10
222220bb
669000b7
69990bb7
669  777
443  111
43388115
4388a555
438aaaa5
```

Figure 22.4

The source code: https://smt.st/current_tree/other/tiling/Z3/tiling.py.
Further reading: https://en.wikipedia.org/wiki/Exact_cover#Pentomino_tiling.

22.7.2 Tiling puzzle (SAT)

This is the same problem, rewritten to be used with SAT solver.

This is different solution: it uses exact cover problem solver which function is in `SAT_lib.py`. (It is better to *offload* it to library.)

The exact cover problem is also used in another example: “Mutually orthogonal Latin squares: finding via transversals”: 12.3.

What is also different: now a polyomino can be reused many times. (This is configurable.)

Also, the *mutilated chessboard problem*[13] can be proven. Of course, no solution would be produced. But SAT solver

[13] https://en.wikipedia.org/wiki/Mutilated_chessboard_problem

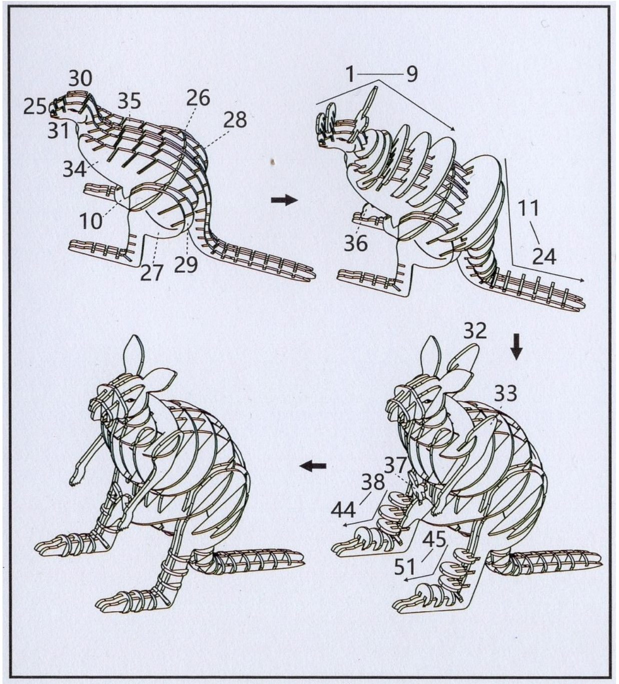
can produce a proof. Resulting CNF file is very compact: 346 clauses, 110 variables. (This is an exact cover problem, of course.) If Kissat is used, the proof file size is only 14KiB.

The source code: https://smt.st/current_tree/other/tiling/SAT/tiling.py.

22.7.3 Kangaroo: Optimizing production of a cardboard toy using SAT-solver

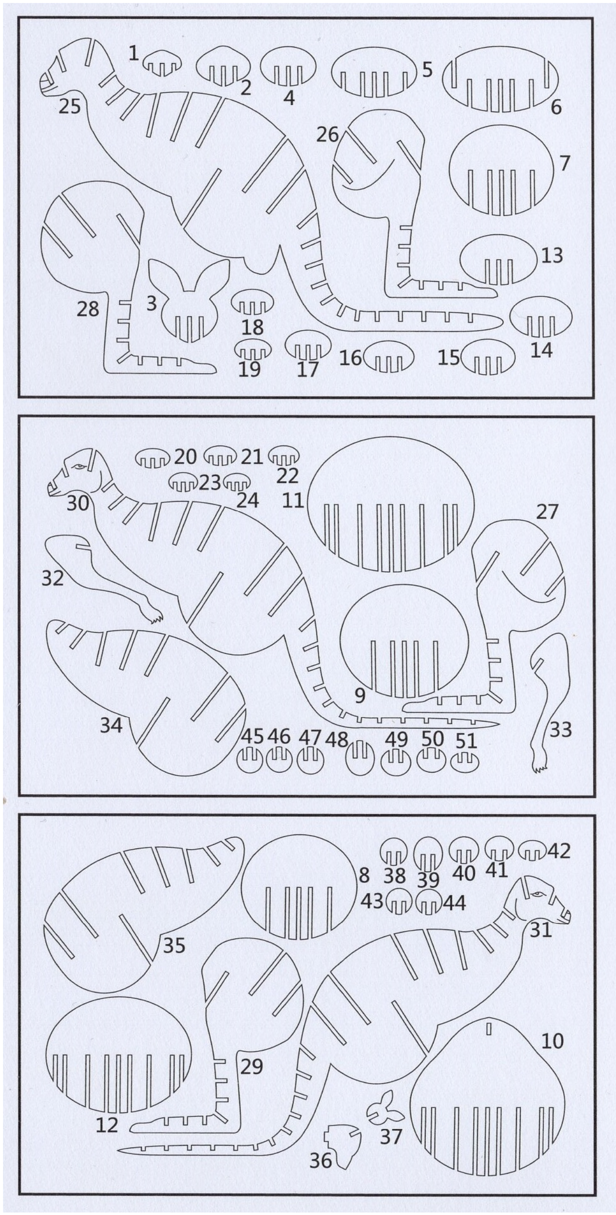
This is a do-it-yourself toy kangaroo I once bought, made of cardboard parts:

(a) Final view

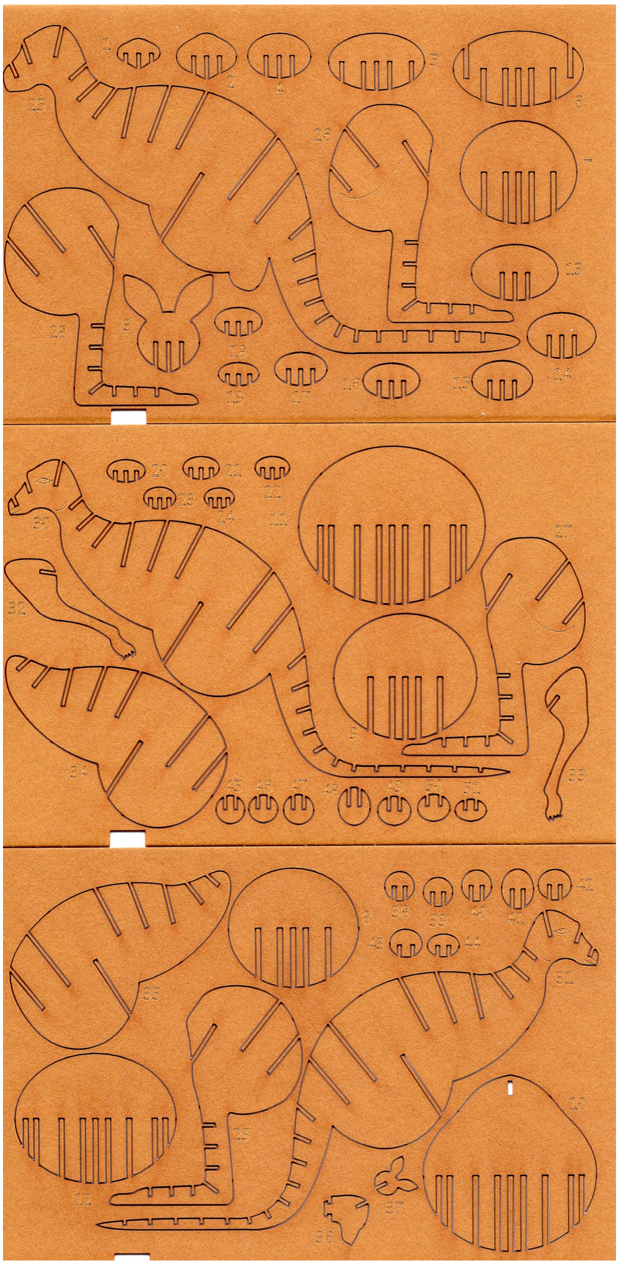


(b) Assembly guide

All parts came in 3 plates:



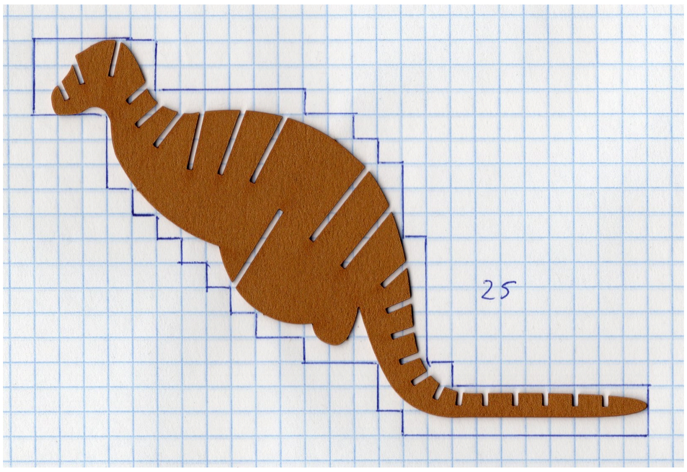
(a) A guide

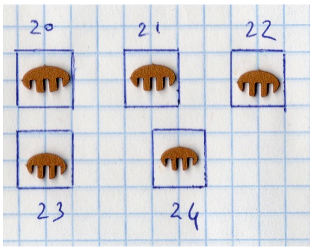


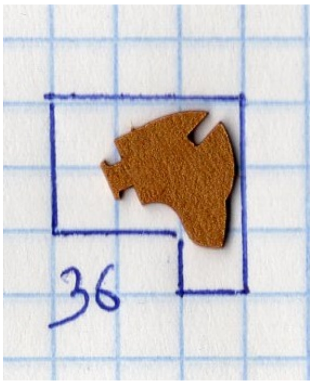
(b) 3 cardboards

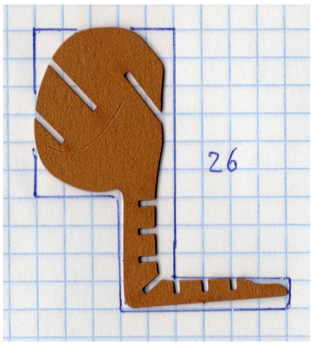
Now the question: can we put all the parts needed on smaller plates? To save some cardboard material?

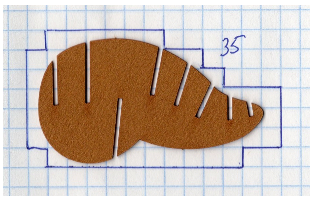
I *digitized* all parts using usual notebook:











I don't know a real size of a square in notebook, probably, 5mm. I would call it *one [square] unit*.

Then I took the same piece of Python code I used before: 22.7.2.

It was easy: there are just (a big) pack of Boolean variables and AMO1/ALO1 constraints, or, as I called them before, POPCNT1.

Thanks to parallelized Plingeling SAT-solver, I could find a solution for a 43*34 [units] plate in 10 minutes[14]:

[14] 4 threads on Intel Xeon CPU E3-1270 v3 @ 3.50GHz

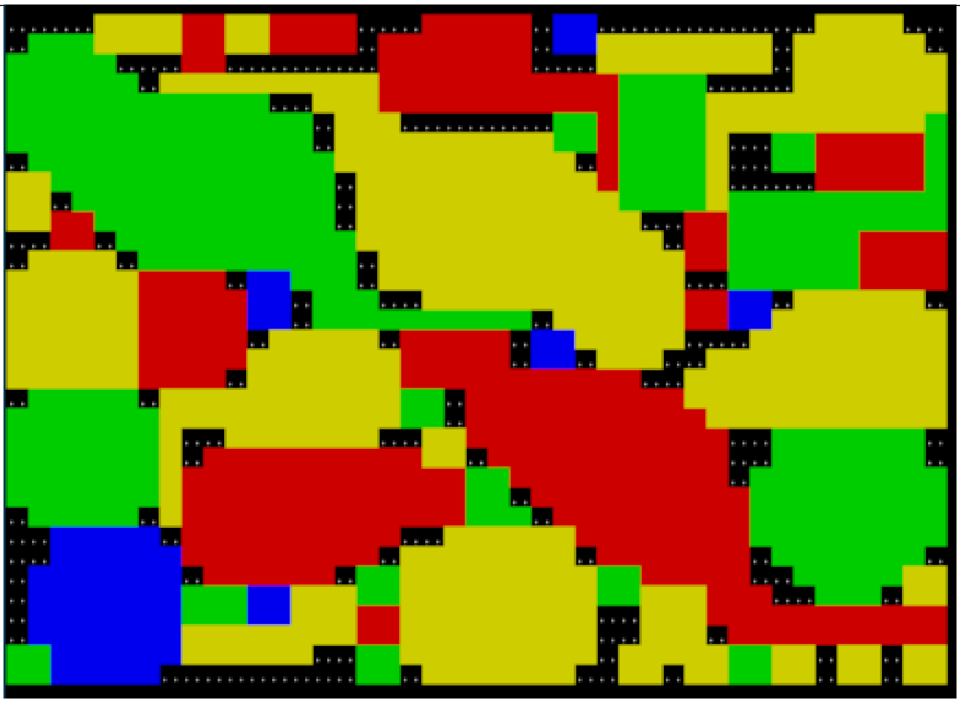


Figure 22.8: All parts on a single 43*34 plate

Probably this is smallest plate possible. When I try smaller dimensions, SAT solver stuck. But if you wish, you can decrease dimensions and run it again and again...

Now the question: the toy factory wants to ship all parts in several (smaller) plates. Like, 3 of them. Because one plate is impractical for shipping, handling, etc.

To put all parts on 3 plates, I can just add 2 borders between them:

```
board=["*"*BOARD_SMALL_WIDTH + " " + "*"*BOARD_SMALL_WIDTH + " " + "*"*
    BOARD_SMALL_WIDTH]*BOARD_SMALL_HEIGHT
```

Smallest (3) plates I found: 19*27 [units]:

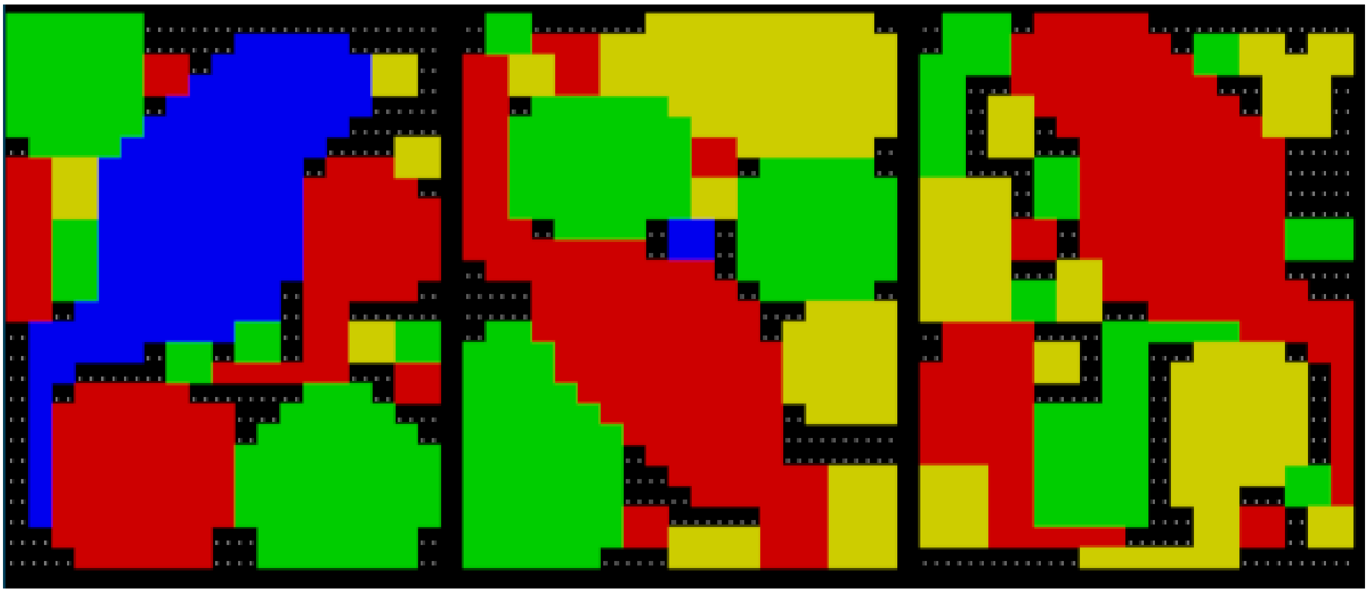


Figure 22.9: All parts on a three 19*27 plates

This is slightly better than what was produced by the toy factory (20*30 [units], as measured by my notebook). Now you can see there are two $2 \cdot 1$ notches at each *plate*:

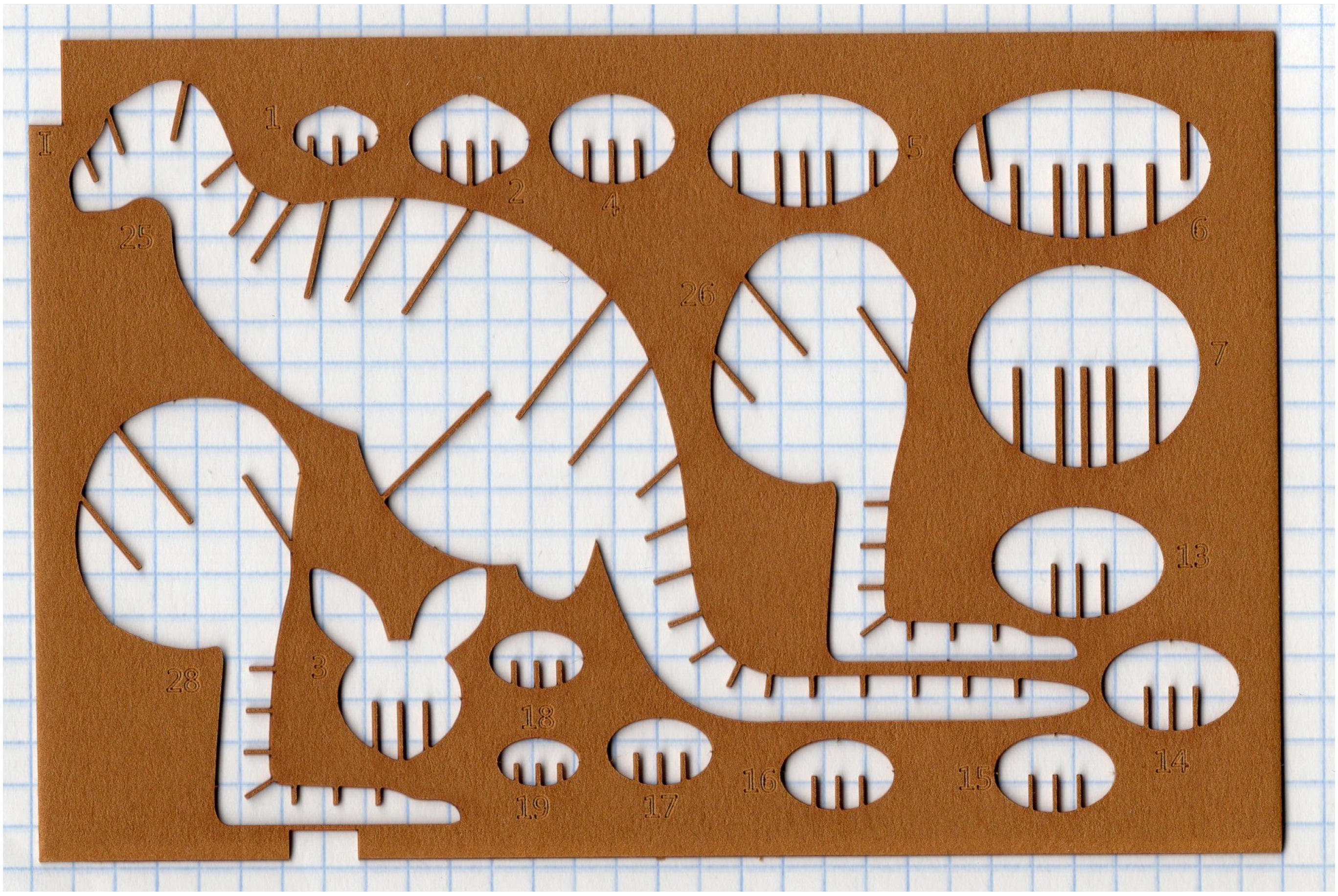


Figure 22.10: Note notches: upper left and bottom left

I don’t know why they been cut. Probably for easier stacking at factory? Who knows. Anyway, I think I can add this constraint to my solver. These are 3 initial plates:

```
board=["*****************   *****************   *****************  ",
       "******************* ******************* *******************",
       "******************* ******************* *******************",
       "******************* ******************* *******************",
       "******************* ******************* *******************",
       "******************* ******************* *******************",
       " ******************  ******************  ******************",
       " ******************  ******************  ******************",
       "******************* ******************* *******************",
       "******************* ******************* *******************",
       "******************* ******************* *******************",
       "******************* ******************* *******************",
       "******************* ******************* *******************",
       "******************* ******************* *******************",
       "******************* ******************* *******************",
       "******************* ******************* *******************",
       "******************* ******************* *******************",
       "******************* ******************* *******************",
       "******************* ******************* *******************",
       "******************* ******************* *******************",
       "******************* ******************* *******************",
       "******************* ******************* *******************",
       "******************* ******************* *******************",
       "******************* ******************* *******************",
       "******************* ******************* *******************",
       "******************* ******************* *******************",
       "******************* ******************* *******************"]
```

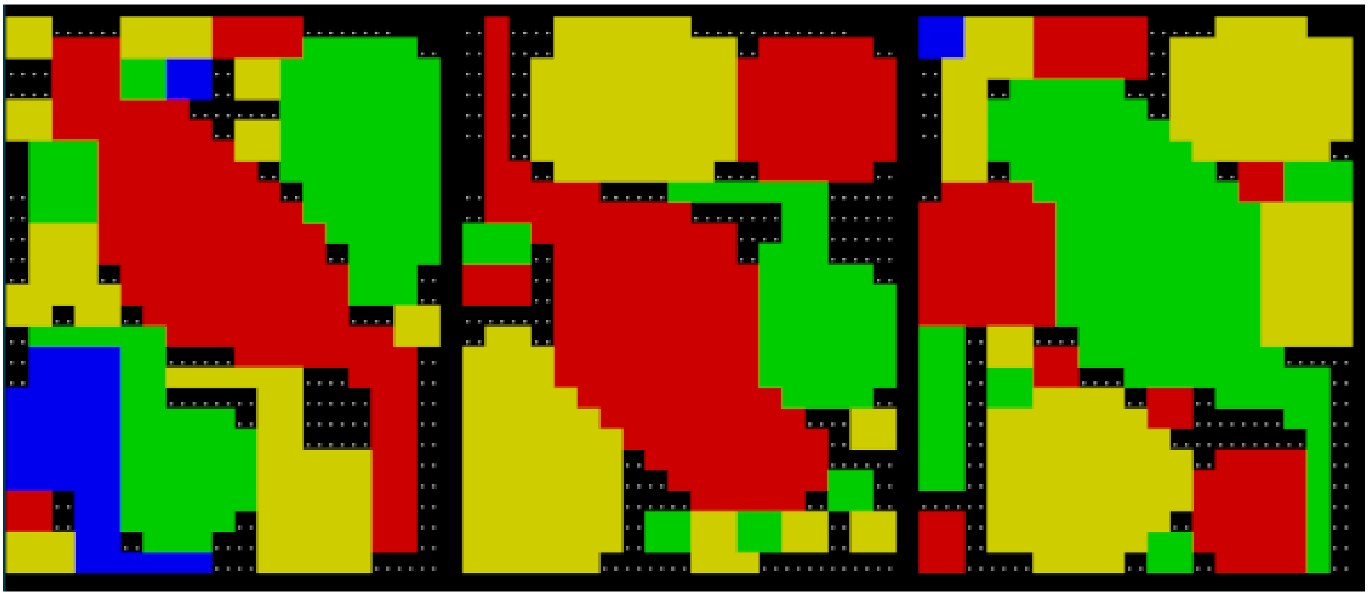


Figure 22.11: All parts on a three 19*27 plates with notches

Keep in mind, how coarse my *units* are (5mm). You can *digitize* better if you use a millimeter paper[15], but such a problem would be more hard for SAT solver, of course.

What I also did: this problem required huge AMO1/ALO1 constraints (several thousands Boolean variables). Naive quadratic encoding can't manage this, also, CNF instances growing greatly.

I used *commander* encoding this time. For example, you need to add AMO1/ALO1 constraint to 100 variables. Divide them by 10 parts. Add naive/quadratic AMO1/ALO1 for each of these 10 parts. Add *OR* for each parts. Then you get 10 *OR result*. Each *OR result* is a *commander*, like, a commander of a squad. Join them together with quadratic AMO1/ALO1 constraint again.

I do this recursively, so it looks like a multi-tiered tree of *commanders*. Also, changing these constants (5 and 10) influences SAT solver's performance significantly, probably, tuning is required for each type of task...

(The constants defines breadth and depth of a tree.)

```
    # naive/pairwise/quadratic encoding
    def AtMost1_pairwise(self, lst:List[str]):
        for pair in itertools.combinations(lst, r=2):
            self.add_clause([self.neg(pair[0]), self.neg(pair[1])])

    # "commander" (?) encoding
    def AtMost1_commander(self, lst:List[str]) -> str:
        parts=my_utils.partition(lst, 5)
        c=[] # type: List[str]
        for part in parts:
            if len(part)<10:
                self.AtMost1_pairwise(part)
                c.append(self.OR_list(part))
            else:
                c.append(self.AtMost1_commander(part))
        self.AtMost1_pairwise(c)
        return self.OR_list(c)

    def AtMost1(self, lst:List[str]):
        if len(lst)<=10:
            self.AtMost1_pairwise(lst)
        else:
            self.AtMost1_commander(lst)

    # previously named POPCNT1
    # make one-hot (AKA unitary) variable
    def make_one_hot(self, lst:List[str]):
```

[15] https://en.wikipedia.org/wiki/Graph_paper

```
        self.AtMost1(lst)
        self.OR_always(lst)
```

(https://smt.st/current_tree/libs/SAT_lib.py)

The files: https://smt.st/current_tree/other/tiling/kangaroo.

22.8 Job Shop Scheduling/Problem

You have number of machines and number of jobs. Each jobs consists of tasks, each task is to be processed on a machine, in specific order.

Probably, this can be a restaurant, each dish is a job. However, a dish is to be cooked in a multi-stage process, and each stage/task require specific kitchen appliance and/or chef. Each appliance/chef at each moment can be busy with only one single task.

The problem is to schedule all jobs/tasks so that they will finish as soon as possible.

See also: https://en.wikipedia.org/wiki/Job_shop_scheduling, https://developers.google.com/optimization/scheduling/job_shop.

22.8.1 My first version

```
#!/usr/bin/env python3
from z3 import *
import itertools

jobs=[]
"""
# from https://developers.google.com/optimization/scheduling/job_shop
jobs.append([(0, 3), (1, 2), (2, 2)])
jobs.append([(0, 2), (2, 1), (1, 4)])
jobs.append([(1, 4), (2, 3)])

machines=3
makespan=11
"""

#"""
# from http://support.sas.com/documentation/cdl/en/orcpug/63973/HTML/default/viewer.htm#orcpug_clp_sect048.htm
jobs.append([(2, 44), (3,  5), (5, 58), (4, 97), (0,  9), (7, 84), (8, 77), (9, 96), (1, 58), (6, 89)])
jobs.append([(4, 15), (7, 31), (1, 87), (8, 57), (0, 77), (3, 85), (2, 81), (5, 39), (9, 73), (6, 21)])
jobs.append([(9, 82), (6, 22), (4, 10), (3, 70), (1, 49), (0, 40), (8, 34), (2, 48), (7, 80), (5, 71)])
jobs.append([(1, 91), (2, 17), (7, 62), (5, 75), (8, 47), (4, 11), (3,  7), (6, 72), (9, 35), (0, 55)])
jobs.append([(6, 71), (1, 90), (3, 75), (0, 64), (2, 94), (8, 15), (4, 12), (7, 67), (9, 20), (5, 50)])
jobs.append([(7, 70), (5, 93), (8, 77), (2, 29), (4, 58), (6, 93), (3, 68), (1, 57), (9,  7), (0, 52)])
jobs.append([(6, 87), (1, 63), (4, 26), (5,  6), (2, 82), (3, 27), (7, 56), (8, 48), (9, 36), (0, 95)])
jobs.append([(0, 36), (5, 15), (8, 41), (9, 78), (3, 76), (6, 84), (4, 30), (7, 76), (2, 36), (1,  8)])
jobs.append([(5, 88), (2, 81), (3, 13), (6, 82), (4, 54), (7, 13), (8, 29), (9, 40), (1, 78), (0, 75)])
jobs.append([(9, 88), (4, 54), (6, 64), (7, 32), (0, 52), (2,  6), (8, 54), (5, 82), (3,  6), (1, 26)])

machines=10
makespan=842
```

```
#"""

# two intervals must not overlap with each other:
def must_not_overlap (s, i1, i2):
    (i1_begin, i1_end)=i1
    (i2_begin, i2_end)=i2
    s.add(Or(i2_begin>=i1_end, i2_begin<i1_begin))
    s.add(Or(i2_end>i1_end, i2_end<=i1_begin))
    (i1_begin, i1_end)=i2
    (i2_begin, i2_end)=i1
    s.add(Or(i2_begin>=i1_end, i2_begin<i1_begin))
    s.add(Or(i2_end>i1_end, i2_end<=i1_begin))

def all_items_in_list_must_not_overlap_each_other(s, lst):
    # enumerate all pairs using Python itertools:
    for pair in itertools.combinations(lst, r=2):
        must_not_overlap(s, (pair[0][1], pair[0][2]), (pair[1][1], pair[1][2]))

s=Solver()

# this is placeholder for tasks, to be indexed by machine number:
tasks_for_machines=[[] for i in range(machines)]

# this is placeholder for jobs, to be indexed by job number:
jobs_array=[]

for job in range(len(jobs)):
    prev_task_end=None
    jobs_array_tmp=[]
    for t in jobs[job]:
        machine=t[0]
        duration=t[1]
        # declare Z3 variables:
        begin=Int('j_%d_task_%d_%d_begin' % (job, machine, duration))
        end=Int('j_%d_task_%d_%d_end' % (job, machine, duration))
        # add variables...
        if (begin,end) not in tasks_for_machines[machine]:
            tasks_for_machines[machine].append((job,begin,end))
        if (begin,end) not in jobs_array_tmp:
            jobs_array_tmp.append((job,begin,end))
        # each task must start at time >= 0
        s.add(begin>=0)
        # end time is fixed with begin time:
        s.add(end==begin+duration)
        # no task must end after makespan:
        s.add(end<=makespan)
        # no task must begin before the end of the last task:
        if prev_task_end!=None:
            s.add(begin>=prev_task_end)
        prev_task_end=end
    jobs_array.append(jobs_array_tmp)

# all tasks on each machine must not overlap each other:
for tasks_for_machine in tasks_for_machines:
    all_items_in_list_must_not_overlap_each_other(s, tasks_for_machine)

# all tasks in each job must not overlap each other:
for jobs_array_tmp in jobs_array:
    all_items_in_list_must_not_overlap_each_other(s, jobs_array_tmp)

if s.check()==unsat:
    print ("unsat")
    exit(0)
m=s.model()

text_result=[]

# construct Gantt chart:
for machine in range(machines):
    st=[None for i in range(makespan)]
    for task in tasks_for_machines[machine]:
        job=task[0]
        begin=m[task[1]].as_long()
        end=m[task[2]].as_long()
        # fill text string with this job number:
        for i in range(begin,end):
            st[i]=job
    ss=""
    for i,t in enumerate(st):
        ss=ss+("." if t==None else str(st[i]))
    text_result.append(ss)

# we need this juggling to rotate Gantt chart...

print ("machines :", end=' ')
for m in range(len(text_result)):
    print (m, end=' ')
print ("")
print ("---------")

for time_unit in range(len(text_result[0])):
    print ("t=%3d    :" % (time_unit), end=' ')
    for m in range(len(text_result)):
        print (text_result[m][time_unit], end=' ')
    print ("")
```

(Syntax-highlighted version: https://smt.st/current_tree/other/job_shop/job.py)

The solution for the 3*3 (3 jobs and 3 machines) problem:

```
machines : 0 1 2
---------
t=  0    : 1 . .
t=  1    : 1 2 .
t=  2    : 0 2 .
t=  3    : 0 2 .
t=  4    : 0 2 1
t=  5    : . 0 2
t=  6    : . 0 2
t=  7    : . 1 2
t=  8    : . 1 0
t=  9    : . 1 0
t= 10    : . 1 .
```

It takes 20s on my venerable Intel Xeon E3-1220 3.10GHz to solve 10*10 (10 jobs and 10 machines) problem from sas.com: https://smt.st/current_tree/other/job_shop/r2.txt.

Further work: makespan can be decreased gradually, or maybe binary search can be used...

22.8.2 MaxSMT version by Chad Brewbaker

Listing 22.8: Diff

```
--- job.py      2021-02-09 03:12:08.008504834 +0200
+++ job_fix.py  2021-02-09 03:14:03.338478978 +0200
```

```
@@ -27,26 +27,23 @@
 jobs.append([(9, 88), (4, 54), (6, 64), (7, 32), (0, 52), (2,  6), (8, 54), (5, 82), (3,  6), (1, 26)])
 
 machines=10
-makespan=842
+#makespan=842
 #"""
+makespan = Int('makespan')
 
 # two intervals must not overlap with each other:
 def must_not_overlap (s, i1, i2):
     (i1_begin, i1_end)=i1
     (i2_begin, i2_end)=i2
-    s.add(Or(i2_begin>=i1_end, i2_begin<i1_begin))
-    s.add(Or(i2_end>i1_end, i2_end<=i1_begin))
-    (i1_begin, i1_end)=i2
-    (i2_begin, i2_end)=i1
-    s.add(Or(i2_begin>=i1_end, i2_begin<i1_begin))
-    s.add(Or(i2_end>i1_end, i2_end<=i1_begin))
+    s.add(Or(i2_begin>=i1_end, i1_begin>=i2_end))
 
 def all_items_in_list_must_not_overlap_each_other(s, lst):
     # enumerate all pairs using Python itertools:
     for pair in itertools.combinations(lst, r=2):
         must_not_overlap(s, (pair[0][1], pair[0][2]), (pair[1][1], pair[1][2]))
 
-s=Solver()
+s = Optimize()
+s.add(makespan>0)
 
 # this is placeholder for tasks, to be indexed by machine number:
 tasks_for_machines=[[] for i in range(machines)]
@@ -88,16 +85,20 @@
 for jobs_array_tmp in jobs_array:
     all_items_in_list_must_not_overlap_each_other(s, jobs_array_tmp)
 
+h = s.minimize(makespan)
+
 if s.check()==unsat:
     print ("unsat")
     exit(0)
+s.lower(h)
 m=s.model()
 
 text_result=[]
 
 # construct Gantt chart:
+ms_long = m[makespan].as_long()
 for machine in range(machines):
-    st=[None for i in range(makespan)]
+    st=[None for i in range(ms_long)]
     for task in tasks_for_machines[machine]:
         job=task[0]
         begin=m[task[1]].as_long()
```

(Full source-code: https://smt.st/current_tree/other/job_shop/job_fix.py)
Works faster, but using MaxSMT.

Chapter 23

Toy-level solvers

... which has been written by me and serves as a demonstration and playground.

23.1 Simplest SAT solver in ~120 lines

This is simplest possible backtracking SAT solver written in Python (not a DPLL[1] one). It uses the same backtracking algorithm you can find in many simple Sudoku and 8 queens solvers. It works significantly slower, but due to its extreme simplicity, it can also count solutions. For example, it can count all solutions of 8 queens problem (8.4).

Also, there are 70 solutions for POPCNT4 function [2] (the function is true if any 4 of its input 8 variables are true):

```
SAT
-1 -2 -3 -4 5 6 7 8 0
SAT
-1 -2 -3 4 -5 6 7 8 0
SAT
-1 -2 -3 4 5 -6 7 8 0
SAT
-1 -2 -3 4 5 6 -7 8 0
...

SAT
1 2 3 -4 -5 6 -7 -8 0
SAT
1 2 3 -4 5 -6 -7 -8 0
SAT
1 2 3 4 -5 -6 -7 -8 0
UNSAT
solutions= 70
```

It was also tested on my SAT-based Minesweeper cracker (3.10.2), and finishes in reasonable time (though, slower than MiniSat by a factor of ~10).

On bigger CNF instances, it gets stuck, though.

The source code:

```
#!/usr/bin/env python3

count_solutions=True
#count_solutions=False

import sys

def read_text_file (fname):
    with open(fname) as f:
        content = f.readlines()
    return [x.strip() for x in content]

```

[1] Davis–Putnam–Logemann–Loveland

[2] https://smt.st/current_tree/solvers/backtrack/POPCNT4.cnf

```
def read_DIMACS (fname):
    content=read_text_file(fname)

    header=content[0].split(" ")

    assert header[0]=="p" and header[1]=="cnf"
    variables_total, clauses_total = int(header[2]), int(header[3])

    # array idx=number (of line) of clause
    # val=list of terms
    # term can be negative signed integer
    clauses=[]
    for c in content[1:]:
        if c.startswith ("c "):
            continue
        clause=[]
        for var_s in c.split(" "):
            var=int(var_s)
            if var!=0:
                clause.append(var)
        clauses.append(clause)

    # this is variables index.
    # for each variable, it has list of clauses, where this variable is used.
    # key=variable
    # val=list of numbers of clause
    variables_idx={}
    for i in range(len(clauses)):
        for term in clauses[i]:
            variables_idx.setdefault(abs(term), []).append(i)

    return clauses, variables_idx

# clause=list of terms. signed integer. -x means negated.
# values=list of values: from 0th: [F,F,F,F,T,F,T,T...]
def eval_clause (terms, values):
    try:
        # we search for at least one True
        for t in terms:
            # variable is non-negated:
            if t>0 and values[t-1]==True:
                return True
            # variable is negated:
            if t<0 and values[(-t)-1]==False:
                return True
        # all terms enumerated at this point
        return False
    except IndexError:
        # values[] has not enough values
        # None means "maybe"
        return None

def chk_vals(clauses, variables_idx, vals):
    # check only clauses which affected by the last (new/changed) value, ignore the
       rest
    # because since we already got here, all other values are correct, so no need to
       recheck them
    idx_of_last_var=len(vals)
    # variable can be absent in index, because no clause uses it:
    if idx_of_last_var not in variables_idx:
        return True
    # enumerate clauses which has this variable:
    for clause_n in variables_idx[idx_of_last_var]:
        clause=clauses[clause_n]
        # if any clause evaluated to False, stop checking, new value is incorrect:
        if eval_clause (clause, vals)==False:
            return False
    # all clauses evaluated to True or None ("maybe")
    return True

def print_vals(vals):
    # enumerate all vals[]
    # prepend "-" if vals[i] is False (i.e., negated).
    print ("".join([["-",""][vals[i]] + str(i+1) + " " for i in range(len(vals))])+"0
       ")

clauses, variables_idx = read_DIMACS(sys.argv[1])

solutions=0

def backtrack(vals):
    global solutions

    if len(vals)==len(variables_idx):
        # we reached end - all values are correct
        print ("SAT")
        print_vals(vals)
        if count_solutions:
            solutions=solutions+1
            # go back, if we need more solutions:
            return
        else:
            exit(10) # as in MiniSat
        return

    for next in [False, True]:
        # add new value:
        new_vals=vals+[next]
        if chk_vals(clauses, variables_idx, new_vals):
            # new value is correct, try add another one:
            backtrack(new_vals)
        else:
            # new value (False) is not correct, now try True (variable flip):
            continue

# try to find all values:
backtrack([])
print ("UNSAT")
if count_solutions:
    print ("solutions=", solutions)
exit(20) # as in MiniSat
```

As you can see, all it does is enumerate all possible solutions, but prunes search tree as early as possible. This is backtracking.

Worst time of execution is EXPTIME.

The files: https://smt.st/current_tree/solvers/backtrack.

Some comments: https://www.reddit.com/r/compsci/comments/6jn3th/simplest_sat_solver_in_120_lines/.

23.2 SAT solvers with watched literals/lists

These are couple of my remakes of Donald Knuth's SAT0W SAT solver[3], CWEB file on his website which is very basic and only serves as a demonstration of watch lists. Read more about it in TAOCP 7.2.2.226.2 (algorithm B, pp. 30-31).

In short:

- A variable is what you see in the DIMACS CNF file. A number. Can be positive or negative. 1234 and −1234 are the same variable.
- Literal is a variable plus sign. 1234 and −1234 are two different literals for the same variable. A clause consists of literals. A CNF instance consists of clauses.
- We need to find an assignment for all SAT variables so that all clauses in CNF would be satisfied.
- We don't need to take into account all variables in each clause. Only one literal in each clause must be true to satisfy the whole CNF instance — keep in mind that fact.
- A literal of our current focus in each clause is called *watched literal* or *watchee*. Each *watchee* is connected to a literal in a database, forming a *watch list*. Default *watchee* is a first literal of a clause.
- *Assignment* is a goal of a SAT solver: a list of false/true variables. *Partial assignment* is an assignment of several variables, not all of them. During solving, each *watchee* is either connected to a literal in a partial assignment, or to (yet) unassigned literal.
- When we switch a variable from false to true (or back) in a partial assignment, a watch list connected to a false literal is to be *disemboweled*: all clauses in watch list are to be reconnected to other literals, either under partial assignment, or *shoved* into yet unassigned literals (in other words, postponed to future). Reconnecting involves finding another *watchee* to be picked from literals in a clause. If you can't find another *watchee*, either switch to another alternative for this variable (false/true) or backtrack.
- Several books and articles says that in this scheme, all clauses are always satisfied, this is like invariant. In my opinion, this is not correct. All clauses connected to watch lists under partial assignment during solving are satisfied, so far. While we can't say this about other clauses connected to watch lists behind partial assignment: they are to be processed in future.
- Essentially, the whole process of SAT solving in this tiny SAT solver is moving clauses from one *watch list* to another.

The Python source code, 190 SLOC

It can solve many tiny SAT instances[4] such as queens on a 10*10 chess board, etc.

Now my second solver in C/C++, which works almost like Donald Knuth's SAT0W. My goal was to remake it precisely, so that I can be sure I understood everything well. To be run fast, there is no recursion.

615 SLOC of C/C++

Also, as we may notice, clauses are not to be added to a watch list or removed. They are rather *moved*. Also, order of clauses in watch list is not important at all. Hence, by moving a clause from one watch list to another, we can add it to the front of destination watch list.

My solution is single-linked lists, but with no pointers. Rather, indices are stored (and -1 is a terminating value). Like Donald Knuth's SAT0W, my solver can even factorize small 8-bit numbers.

Needless to say, an order of variables influences the process drastically. Hence, my solver can behave differently if it reads DIMACS CNF file or Knuth-style SAT file (where variable names are used instead of numbers). However, finding a best order of variables is a problem comparable to SAL solving itself.

Also, here is reworked C-source code of D.Knuth's sat0w.w program:

480 SLOC of low-level D.Knuth's pure C

I used it for testing, etc...

23.3 MK85 toy-level SMT-solver

Thanks to PicoSAT SAT solver, its performance on small and simple bitvector examples is comparable to Z3. In many cases, it's used instead of Z3, whenever you see `from MK85 import *` in Python file.

[3] https://smt.st/current_tree/solvers/SAT_WL/sat0w.pdf

[4] https://smt.st/current_tree/solvers/SAT_WL/tests/tests

23.3.1 Simple adder in SAT/SMT

Let's solve the following equation $a + b = 4 \equiv 2^4$ on the 4-bit CPU (hence, modulo 2^4):

```
(declare-fun a () (_ BitVec 4))
(declare-fun b () (_ BitVec 4))

(assert (= (bvadd a b) #x4))

; find a, b:
(get-all-models)
```

There are 16 possible solutions (easy to check even by hand):

```
(model
        (define-fun a () (_ BitVec 4) (_ bv0 4)) ; 0x0
        (define-fun b () (_ BitVec 4) (_ bv4 4)) ; 0x4
)
(model
        (define-fun a () (_ BitVec 4) (_ bv12 4)) ; 0xc
        (define-fun b () (_ BitVec 4) (_ bv8 4)) ; 0x8
)

...

(model
        (define-fun a () (_ BitVec 4) (_ bv9 4)) ; 0x9
        (define-fun b () (_ BitVec 4) (_ bv11 4)) ; 0xb
)
Model count: 16
```

How I implemented this in my toy-level SMT solver?

First, we need an electronic adder, like it's implemented in digital circuits. Take a look at a full adder: 3.7.

And this is how full adders gathered together to form a simple 4-bit carry-ripple adder: 3.9.

More info from wikipedia: https://en.wikipedia.org/wiki/Adder_(electronics).

I'm implementing full-adder like this:

```
void add_Tseitin_XOR(int v1, int v2, int v3)
{
        add_comment ("%s %d=%d^%d", __FUNCTION__, v3, v1, v2);
        add_clause3 (-v1, -v2, -v3);
        add_clause3 (v1, v2, -v3);
        add_clause3 (v1, -v2, v3);
        add_clause3 (-v1, v2, v3);
};

void add_Tseitin_OR2(int v1, int v2, int var_out)
{
        add_comment ("%s %d=%d|%d", __FUNCTION__, var_out, v1, v2);
        add_clause("%d %d -%d", v1, v2, var_out);
        add_clause2(-v1, var_out);
        add_clause2(-v2, var_out);
};

void add_Tseitin_AND(int a, int b, int out)
{
        add_comment ("%s %d=%d&%d", __FUNCTION__, out, a, b);
        add_clause3 (-a, -b, out);
        add_clause2 (a, -out);
        add_clause2 (b, -out);
};

void add_FA(int a, int b, int cin, int s, int cout)
```

```
{
        add_comment("%s inputs=%d, %d, cin=%d, s=%d, cout=%d", __FUNCTION__, a, b,
            cin, s, cout);
        // allocate 3 "joint" variables:
        int XOR1_out=next_var_no++;
        int AND1_out=next_var_no++;
        int AND2_out=next_var_no++;
        // add gates and connect them.
        // order doesn't matter, BTW:
        add_Tseitin_XOR(a, b, XOR1_out);
        add_Tseitin_XOR(XOR1_out, cin, s);
        add_Tseitin_AND(XOR1_out, cin, AND1_out);
        add_Tseitin_AND(a, b, AND2_out);
        add_Tseitin_OR2(AND1_out, AND2_out, cout);
};
```

(https://smt.st/MK85)

|add_Tseitin*()| functions makes logic gates in CNF form: https://en.wikipedia.org/wiki/Tseytin_transformation. Then I connect logic gates to make full-adder.

Then I connect full-adders to create a n-bit adder:

```
void generate_adder(struct variable* a, struct variable* b, struct variable *carry_in
    , // inputs
        struct variable** sum, struct variable** carry_out) // outputs
{

        ...

        *sum=create_internal_variable("adder_sum", TY_BITVEC, a->width);

        int carry=carry_in->var_no;

        // the first full-adder could be half-adder, but we make things simple here
        for (int i=0; i<a->width; i++)
        {
                *carry_out=create_internal_variable("adder_carry", TY_BOOL, 1);
                add_FA(a->var_no+i, b->var_no+i, carry, (*sum)->var_no+i, (*carry_out
                    )->var_no);
                // newly created carry_out is a carry_in for the next full-adder:
                carry=(*carry_out)->var_no;
        };
};
```

(https://smt.st/MK85)

Let's take a look on output CNF file:

```
p cnf 40 114
c always false
-1 0
c always true
2 0
c generate_adder
c add_FA inputs=3, 7, cin=1, s=11, cout=15
c add_Tseitin_XOR 16=3^7
-3 -7 -16 0
3 7 -16 0
3 -7 16 0
-3 7 16 0
c add_Tseitin_XOR 11=16^1
-16 -1 -11 0
16 1 -11 0
16 -1 11 0
-16 1 11 0
```

```
c add_Tseitin_AND 17=16&1
-16 -1 17 0
16 -17 0
1 -17 0
c add_Tseitin_AND 18=3&7
-3 -7 18 0
3 -18 0
7 -18 0
c add_Tseitin_OR2 15=17|18
17 18 -15 0
-17 15 0
-18 15 0
c add_FA inputs=4, 8, cin=15, s=12, cout=19
c add_Tseitin_XOR 20=4^8
-4 -8 -20 0
4 8 -20 0
4 -8 20 0
-4 8 20 0
c add_Tseitin_XOR 12=20^15
-20 -15 -12 0
20 15 -12 0
20 -15 12 0
-20 15 12 0
c add_Tseitin_AND 21=20&15
-20 -15 21 0
20 -21 0
15 -21 0
c add_Tseitin_AND 22=4&8
-4 -8 22 0
4 -22 0
8 -22 0
c add_Tseitin_OR2 19=21|22
21 22 -19 0
-21 19 0
-22 19 0
c add_FA inputs=5, 9, cin=19, s=13, cout=23
c add_Tseitin_XOR 24=5^9
-5 -9 -24 0
5 9 -24 0
5 -9 24 0
-5 9 24 0
c add_Tseitin_XOR 13=24^19
-24 -19 -13 0
24 19 -13 0
24 -19 13 0
-24 19 13 0
c add_Tseitin_AND 25=24&19
-24 -19 25 0
24 -25 0
19 -25 0
c add_Tseitin_AND 26=5&9
-5 -9 26 0
5 -26 0
9 -26 0
c add_Tseitin_OR2 23=25|26
25 26 -23 0
-25 23 0
-26 23 0
c add_FA inputs=6, 10, cin=23, s=14, cout=27
c add_Tseitin_XOR 28=6^10
-6 -10 -28 0
```

```
6 10 -28 0
6 -10 28 0
-6 10 28 0
c add_Tseitin_XOR 14=28^23
-28 -23 -14 0
28 23 -14 0
28 -23 14 0
-28 23 14 0
c add_Tseitin_AND 29=28&23
-28 -23 29 0
28 -29 0
23 -29 0
c add_Tseitin_AND 30=6&10
-6 -10 30 0
6 -30 0
10 -30 0
c add_Tseitin_OR2 27=29|30
29 30 -27 0
-29 27 0
-30 27 0
c generate_const(val=4, width=4). var_no=[31..34]
-31 0
-32 0
33 0
-34 0
c generate_EQ for two bitvectors, v1=[11...14], v2=[31...34]
c generate_BVXOR v1=[11...14] v2=[31...34]
c add_Tseitin_XOR 35=11^31
-11 -31 -35 0
11 31 -35 0
11 -31 35 0
-11 31 35 0
c add_Tseitin_XOR 36=12^32
-12 -32 -36 0
12 32 -36 0
12 -32 36 0
-12 32 36 0
c add_Tseitin_XOR 37=13^33
-13 -33 -37 0
13 33 -37 0
13 -33 37 0
-13 33 37 0
c add_Tseitin_XOR 38=14^34
-14 -34 -38 0
14 34 -38 0
14 -34 38 0
-14 34 38 0
c generate_OR_list(var=35, width=4) var out=39
c add_Tseitin_OR_list(var=35, width=4, var_out=39)
35 36 37 38 -39 0
-35 39 0
-36 39 0
-37 39 0
-38 39 0
c generate_NOT id=internal!8 var=39, out id=internal!9 out var=40
-40 -39 0
40 39 0
c create_assert() id=internal!9 var=40
40 0
```

Filter out comments:

```
c always false
c always true
c generate_adder
c add_FA inputs=3, 7, cin=1, s=11, cout=15
c add_Tseitin_XOR 16=3^7
c add_Tseitin_XOR 11=16^1
c add_Tseitin_AND 17=16&1
c add_Tseitin_AND 18=3&7
c add_Tseitin_OR2 15=17|18
c add_FA inputs=4, 8, cin=15, s=12, cout=19
c add_Tseitin_XOR 20=4^8
c add_Tseitin_XOR 12=20^15
c add_Tseitin_AND 21=20&15
c add_Tseitin_AND 22=4&8
c add_Tseitin_OR2 19=21|22
c add_FA inputs=5, 9, cin=19, s=13, cout=23
c add_Tseitin_XOR 24=5^9
c add_Tseitin_XOR 13=24^19
c add_Tseitin_AND 25=24&19
c add_Tseitin_AND 26=5&9
c add_Tseitin_OR2 23=25|26
c add_FA inputs=6, 10, cin=23, s=14, cout=27
c add_Tseitin_XOR 28=6^10
c add_Tseitin_XOR 14=28^23
c add_Tseitin_AND 29=28&23
c add_Tseitin_AND 30=6&10
c add_Tseitin_OR2 27=29|30
c generate_const(val=4, width=4). var_no=[31..34]
c generate_EQ for two bitvectors, v1=[11...14], v2=[31...34]
c generate_BVXOR v1=[11...14] v2=[31...34]
c add_Tseitin_XOR 35=11^31
c add_Tseitin_XOR 36=12^32
c add_Tseitin_XOR 37=13^33
c add_Tseitin_XOR 38=14^34
c generate_OR_list(var=35, width=4) var out=39
c add_Tseitin_OR_list(var=35, width=4, var_out=39)
c generate_NOT id=internal!8 var=39, out id=internal!9 out var=40
c create_assert() id=internal!9 var=40
```

I make these functions add variable numbers to comments. And you can see how all the signals are routed inside each full-adder.

|generate_EQ()| function makes two bitvectors equal by XOR-ing two bitvectors. Resulting bitvector is then OR-ed, and result must be zero.

Again, this SAT instance is small enough to be handled by my simple SAT backtracking solver:

```
SAT
-1 2 -3 -4 -5 -6 -7 -8 9 -10 -11 -12 13 -14 -15 -16 -17 -18 -19 -20 -21 -22 -23 24
    -25 -26 -27 -28 -29 -30 -31 -32 33 -34 -35 -36
-37 -38 -39 40 0
SAT
-1 2 -3 -4 -5 6 -7 -8 9 10 -11 -12 13 -14 -15 -16 -17 -18 -19 -20 -21 -22 -23 24 -25
    -26 27 -28 -29 30 -31 -32 33 -34 -35 -36 -37
-38 -39 40 0

...

SAT
-1 2 3 4 5 -6 7 -8 9 10 -11 -12 13 -14 15 -16 -17 18 19 20 21 -22 23 -24 -25 26 27 28
     29 -30 -31 -32 33 -34 -35 -36 -37 -38 -39 40 0
SAT
-1 2 3 4 5 6 7 -8 9 -10 -11 -12 13 -14 15 -16 -17 18 19 20 21 -22 23 -24 -25 26 27 28
```

```
    29 -30 -31 -32 33 -34 -35 -36 -37 -38 -39 40 0
UNSAT
solutions= 16
```

23.3.2 Combinatorial optimization

This is minimize/maximize commands in SMT-LIB. See simple example on GCD: 14.4.1.

It was surprisingly easy to add support of it to MK85. First, we take MaxSAT/WBO solver Open-WBO[5]. It supports both hard and soft clauses. Hard are clauses which are *must* be satisfied. Soft are *should* be satisfied, but they are also weighted. The task of MaxSAT solver is to find such an assignment for variables, so the sum of weights of soft clauses would be *maximized*.

This is GCD example rewritten to SMT-LIB format:

```
; checked with Z3 and MK85

; must be 21
; see also: https://www.wolframalpha.com/input/?i=GCD[861,3969,840]

(declare-fun x () (_ BitVec 16))
(declare-fun y () (_ BitVec 16))
(declare-fun z () (_ BitVec 16))
(declare-fun GCD () (_ BitVec 16))

(assert (= (bvmul ((_ zero_extend 16) x) ((_ zero_extend 16) GCD)) (_ bv861 32)))
(assert (= (bvmul ((_ zero_extend 16) y) ((_ zero_extend 16) GCD)) (_ bv3969 32)))
(assert (= (bvmul ((_ zero_extend 16) z) ((_ zero_extend 16) GCD)) (_ bv840 32)))

(maximize GCD)

(check-sat)
(get-model)

; correct result:
;(model
;         (define-fun x () (_ BitVec 16) (_ bv41 16)) ; 0x29
;         (define-fun y () (_ BitVec 16) (_ bv189 16)) ; 0xbd
;         (define-fun z () (_ BitVec 16) (_ bv40 16)) ; 0x28
;         (define-fun GCD () (_ BitVec 16) (_ bv21 16)) ; 0x15
;)
```

We are going to find such an assignment, for which GCD variable will be as big as possible (that would not break *hard* constraints, of course).

Whenever my MK85 encounters *minimize/maximize* command, the following function is called:

```
void create_min_max (struct expr* e, bool min_max)
{
        ...

        struct SMT_var* v=generate(e);

        // if "minimize", negate input value:
        if (min_max==false)
                v=generate_BVNEG(v);

        assert (v->type==TY_BITVEC);
        add_comment ("%s(min_max=%d) id=%s var=%d", __FUNCTION__, min_max, v->id, v->
            SAT_var);

        // maximize always. if we need to minimize, $v$ is negated at this point:
        for (int i=0; i<v->width; i++)
```

[5] http://sat.inesc-id.pt/open-wbo/

```
                add_soft_clause1(/* weight */ 1<<i, v->SAT_var+i);

        ...
};
```

(https://smt.st/MK85)

Lowest bit of variable to maximize receives weight 1. Second bit receives weight 2. Then 4, 8, 16, etc. Hence, MaxSAT solver, in order to maximize weights of soft clauses, would maximize the binary variable as well!

What is in the WCNF file for the GCD example?

```
...

c create_min_max(min_max=1) id=GCD var=51
1 51 0
2 52 0
4 53 0
8 54 0
16 55 0
32 56 0
64 57 0
128 58 0
256 59 0
512 60 0
1024 61 0
2048 62 0
4096 63 0
8192 64 0
16384 65 0
32768 66 0
```

Weights from 1 to 32768 to be assigned to specific bits of GCD variable.

Minimization works just as the same, but the input value is negated.

Now some practical examples MK85 can already solve: Assignment problem: 14.5, Finding minimum of function: 14.9, Minimizing cost10.1.

23.3.3 Making (almost) barrel shifter in my toy-level SMT solver

...so the functions `bvshl` and `bvlshr` (logical shift right) would be supported.

We will simulate barrel shifter, a device which can shift a value by several bits in one cycle.

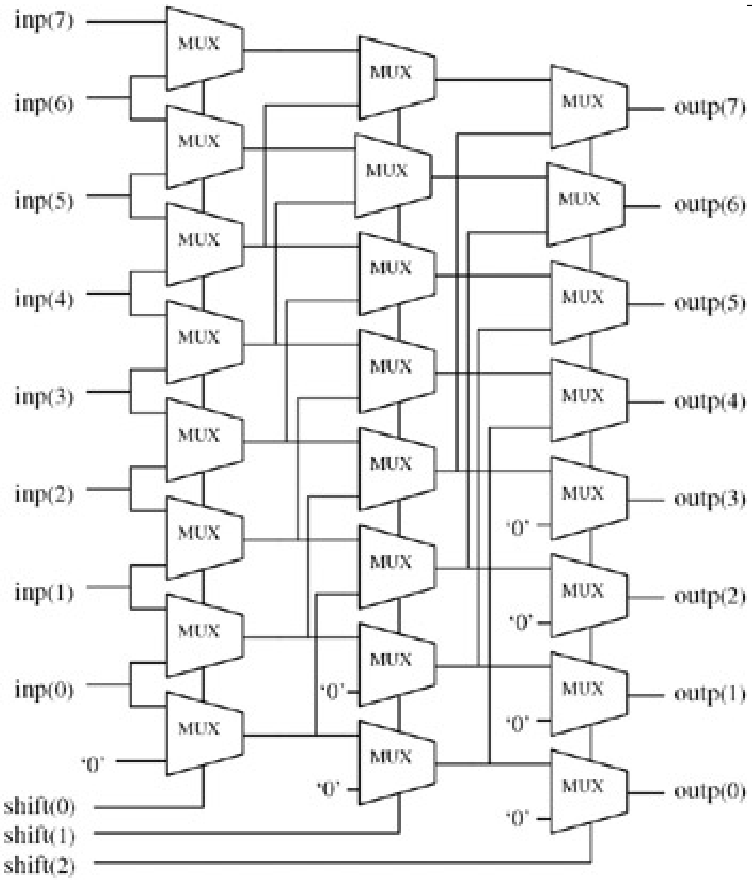


Figure 23.1: A nice illustration of barrel shifter

See also: https://en.wikipedia.org/wiki/Barrel_shifter

So we have a pack of multiplexers. A tier of them for each bit in *cnt* variable (number of bits to shift).

First, I define functions which do *rewiring* rather than shifting, it's just another name. Part of input is *connected* to output bits, other bits are fixed to zero:

```
        // "cnt" is not a SMT variable!
        struct SMT_var* generate_shift_left(struct SMT_var* X, unsigned int cnt)
        {
                int w=X->width;

                struct SMT_var* rt=create_internal_variable("shifted_left", TY_BITVEC
                    , w);

                fix_BV_to_zero(rt->SAT_var, cnt);

                add_Tseitin_EQ_bitvecs(w-cnt, rt->SAT_var+cnt, X->SAT_var);

                return rt;
        };

        // cnt is not a SMT variable!
        struct SMT_var* generate_shift_right(struct SMT_var* X, unsigned int cnt)
        {
                ... likewise
        };
```

It can be said, the |cnt| variable would be set during SAT instance creation, but it cannot be changed during solving.

Now let's create a *real* shifter. Now for 8-bit left shifter, I'm generating the following (long) expression:

```
        X=ITE(cnt&1, X<<1, X)
        X=ITE((cnt>>1)&1, X<<2, X)
```

```
X=ITE((cnt>>2)&1, X<<4, X)
```

I.e., if a specific bit is set in |cnt|, shift X by that number of bits, or do nothing otherwise. ITE() is a if-then-else gate, works for bitvectors as well.

Glueing all this together:

```
// direction=false for shift left
// direction=true for shift right
struct SMT_var* generate_shifter (struct SMT_var* X, struct SMT_var* cnt,
    bool direction)
{
        int w=X->width;

        struct SMT_var* in=X;
        struct SMT_var* out;
        struct SMT_var* tmp;

        // bit vector must have width=2^x, i.e., 8, 16, 32, 64, etc
        assert (popcount64c (w)==1);

        int bits_in_selector=mylog2(w);

        for (int i=0; i<bits_in_selector; i++)
        {
                if (direction==false)
                        tmp=generate_shift_left(in, 1<<i);
                else
                        tmp=generate_shift_right(in, 1<<i);

                out=create_internal_variable("tmp", TY_BITVEC, w);

                add_Tseitin_ITE_BV (cnt->SAT_var+i, tmp->SAT_var, in->SAT_var
                    , out->SAT_var, w);

                in=out;
        };

        // if any bit is set in high part of "cnt" variable, result is 0
        // i.e., if a 8-bit bitvector is shifted by cnt>8, give a zero
        struct SMT_var *disable_shifter=create_internal_variable("...",
            TY_BOOL, 1);
        add_Tseitin_OR_list(cnt->SAT_var+bits_in_selector, w-bits_in_selector
            , disable_shifter->SAT_var);

        return generate_ITE(disable_shifter, generate_const(0, w), in);
};

struct SMT_var* generate_BVSHL (struct SMT_var* X, struct SMT_var* cnt)
{
        return generate_shifter (X, cnt, false);
};
```

Now the puzzle. a>>b must be equal to 0x12345678, while several bits in a must be reset, like (a&0xf1110100)==0. Find a, b:

```
(declare-fun a () (_ BitVec 32))
(declare-fun b () (_ BitVec 32))

(assert (= (bvand a #xf1110100) #x00000000))

(assert (= (bvshl a b) #x12345678))

(check-sat)
```

```
        (get-model)
```

The solution:

```
        sat
        (model
                (define-fun a () (_ BitVec 32) (_ bv38177487 32)) ; 0x2468acf
                (define-fun b () (_ BitVec 32) (_ bv3 32)) ; 0x3
        )
```

23.3.4 A poor man's MaxSMT

First, what is incremental SAT? When a SAT-solver is *warmed up*, and you want only to alter list of clauses slightly by adding one, why to reset it? In my toy SMT solver, I use incremental SAT for model counting, or getting all models: see the picosat_get_all_models() function.

Also, many SAT solvers supports *assumptions*. That is you supply a list of clauses + *temporary* clauses that will be dropped each time.

Now a digression: this is my own binary search implementation:

```
/*

This is yet another explanation of binary search.
However, it works only on array sizes = 2^n

You know, like bank robbers in movies rotating a wheel on a safe (I don't know how it's called correctly)
and find all digits consecutively.
This is like brute-fource.

We do here the same: we try 0/1 for each bit of index value.
We start at 1, and if the array value at this index is too large, we clear the bit to 0 and proceed to the next (lower) bit.

The array here has 32 numbers. The array index has 5 bits ( log2(32)==5 ).
And you can clearly see that one need only 5 steps to find a value in an array of sorted numbers.

The result:

==========================================
testing idx=0x10 or 16
bit 4 is incorrect, it's 0, so we clear it
testing idx=0x8 or 8
bit 3 is correct, it's 1
testing idx=0xc or 12
bit 2 is correct, it's 1
testing idx=0xe or 14
bit 1 is incorrect, it's 0, so we clear it
testing idx=0xd or 13
found, idx=0xd or 13
==========================================

*/

#include <stdlib.h>
#include <stdio.h>

// array of sorted random numbers:
int array[32]=
{
        335, 481, 668, 1169, 1288, 1437, 1485, 1523,
        1839, 2058, 2537, 2585, 2698, 2722, 3245, 3675,
        4067, 4139, 4356, 4599, 5578, 6334, 6751, 7244,
        7845, 8220, 8272, 8296, 8297, 8358, 9138, 9650
};

void binsearch (int val_to_find)
{
        int idx=0;

        for (int bit=4; ; bit--)
        {
                // set the bit:
                idx|=1<<bit;

                printf ("testing idx=0x%x or %d\n", idx, idx);

                if (array[idx]==val_to_find)
                {
                        printf ("found, idx=0x%x or %d\n", idx, idx);
                        exit(0);
                };

                // array[idx] is too small?
                if (array[idx]<val_to_find)
                {
                        // do nothing, the current bit correct, proceed to the next bit
                        printf ("bit %d is correct, it's 1\n", bit);
                };

                // array[idx] is too big?
                if (array[idx]>val_to_find)
                {
                        // clear the current bit, because it's incorrect
                        idx&=~(1<<bit);
                        printf ("bit %d is incorrect, it's 0, so we clear it\n", bit);
                };
        };
};

int main()
{
        binsearch(2722);
};
```

You wouldn't use it, but I wrote it because it's fits so nicely on SAT.

To solve an optimization problem, you want to find some *optimum* variable, that is minimized or maximized. Like in my implementation of binary search, I can try bit by bit, but these are added as assumptions.

```
// poor man's MaxSMT
// if we minimize, first try false, then true
// if we maximize, do contrariwise
bool run_poor_mans_MaxSMT(struct ctx* ctx)
{
        if (verbose>0)
                printf ("%s() begin\n", __FUNCTION__);
        assert (ctx->maxsat==true);
        struct PicoSAT *p=picosat_init ();

        add_clauses_to_picosat(ctx, p);

        if (ctx->write_CNF_file)
        {
                write_CNF(ctx, "tmp.cnf");
                printf ("CNF file written to tmp.cnf\n");
        };

        int array[ctx->min_max_var->width];
        int res;
        // do this starting at the MSB moving towards the LSB:
        for (int idx=ctx->min_max_var->width-1; idx>=0; idx--)
        {
                // try the first false/true
                array[idx]=ctx->min_or_max==false ? 0 : 1;
                if (verbose>0)
                        printf ("idx=%d trying %s\n", idx, ctx->min_or_max==false ? "
                           false" : "true");
                picosat_assume (p, (ctx->min_or_max==false ? -1 : 1) * (ctx->
                   min_max_var->SAT_var+idx));

                res=picosat_sat (p,-1);
                if (res==10)
                {
                        if (verbose>0)
                                printf ("got SAT\n");
                        if (idx!=0)
                        {
                                // add a newly discovered correct bit as a clause
                                // but do this only if this is not the last bit
                                // if the bit is the last/lowest (idx==0), we want to
                                    prevent PicoSAT to be switched out of SAT state
                                // (PicoSAT do this after picosat_add() call)
                                // because we are fetching result sooner afterwards
                                picosat_add (p, (ctx->min_or_max==false ? -1 : 1) * (
                                   ctx->min_max_var->SAT_var+idx));
                                picosat_add (p, 0);
                        }
                        // proceed to the next bit:
                        continue;
                };
                if (verbose>0)
                        printf ("got UNSAT\n");
                assert (res==20); // must be in UNSAT state

                // try the second false/true
                array[idx]=ctx->min_or_max==false ? 1 : 0;
                if (verbose>0)
                        printf ("idx=%d trying %s\n", idx, ctx->min_or_max==false ? "
                           true" : "false");
                picosat_assume (p, (ctx->min_or_max==false ? 1 : -1) * (ctx->
                   min_max_var->SAT_var+idx));

                res=picosat_sat (p,-1);
                if (res==10)
                {
                        if (verbose>0)
                                printf ("got SAT\n");
                        if (idx!=0)
                        {
                                picosat_add (p, (ctx->min_or_max==false ? 1 : -1) * (
                                   ctx->min_max_var->SAT_var+idx));
                                picosat_add (p, 0);
                        };
                }
                else if (res==20)
                {
                        if (verbose>0)
                        {
                                printf ("got UNSAT\n");
                                printf ("%s() begin -> false\n", __FUNCTION__);
                        };
                        // UNSAT for both false and true for this bit at idx, return
                           UNSAT for this instance:
                        return false;
                }
                else
                {
                        assert(0);
                };
        };
        // must have a solution at this point
        fill_variables_from_picosat(ctx, p);
        // construct a value from array[]:
        ctx->min_max_var->val=SAT_solution_to_value(array, ctx->min_max_var->width);
        if (verbose>0)
                printf ("%s() begin -> true, val=%llu\n", __FUNCTION__, ctx->
                   min_max_var->val);
        return true;
};
```

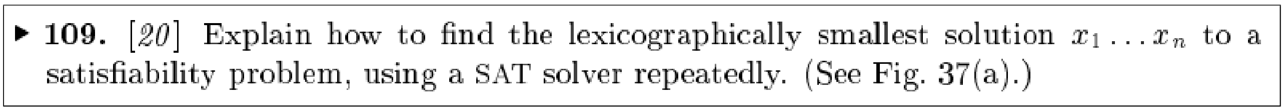
(https://smt.st/MK85)

For the popsicle problem (10.1):

```
idx=15 trying true
got UNSAT
idx=15 trying false
got SAT
idx=14 trying true
got UNSAT
idx=14 trying false
got SAT
idx=13 trying true
got UNSAT
idx=13 trying false
got SAT
idx=12 trying true
got UNSAT
idx=12 trying false
got SAT
idx=11 trying true
got UNSAT
idx=11 trying false
got SAT
idx=10 trying true
got UNSAT
idx=10 trying false
got SAT
idx=9 trying true
got UNSAT
idx=9 trying false
got SAT
idx=8 trying true
got UNSAT
idx=8 trying false
got SAT
```

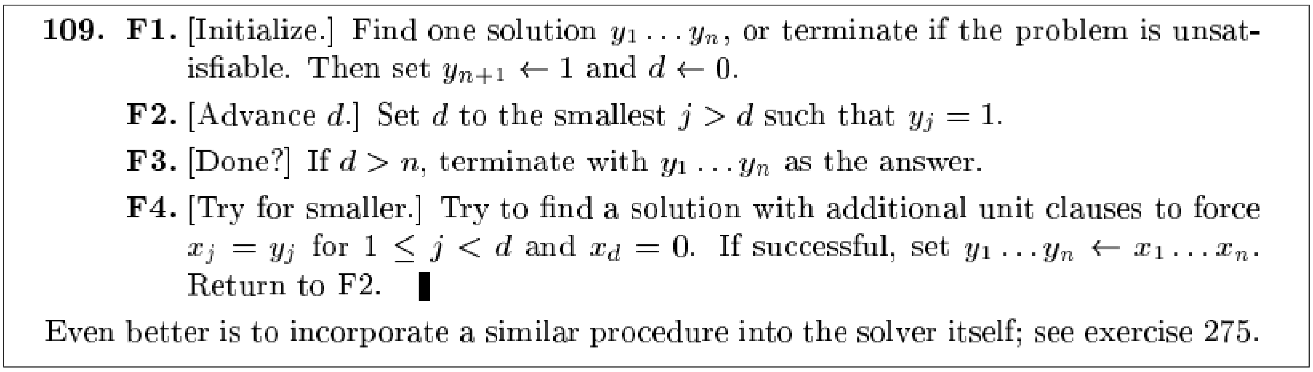
```
idx=7 trying true
got UNSAT
idx=7 trying false
got SAT
idx=6 trying true
got UNSAT
idx=6 trying false
got SAT
idx=5 trying true
got UNSAT
idx=5 trying false
got SAT
idx=4 trying true
got UNSAT
idx=4 trying false
got SAT
idx=3 trying true
got SAT
idx=2 trying true
got SAT
idx=1 trying true
got UNSAT
idx=1 trying false
got SAT
idx=0 trying true
got SAT
...
run_poor_mans_MaxSMT() begin -> true, val=13
```

It works slower than if using Open-WBO, but sometimes even faster than Z3!
This is as well: 14.9.
And this is close to LEXSAT, see exercise and solution from the TAOCP, section 7.2.2.2:



▸ **109.** [*20*] Explain how to find the lexicographically smallest solution $x_1 \ldots x_n$ to a satisfiability problem, using a SAT solver repeatedly. (See Fig. 37(a).)

Figure 23.2: The exercise



109. F1. [Initialize.] Find one solution $y_1 \ldots y_n$, or terminate if the problem is unsatisfiable. Then set $y_{n+1} \leftarrow 1$ and $d \leftarrow 0$.

F2. [Advance d.] Set d to the smallest $j > d$ such that $y_j = 1$.

F3. [Done?] If $d > n$, terminate with $y_1 \ldots y_n$ as the answer.

F4. [Try for smaller.] Try to find a solution with additional unit clauses to force $x_j = y_j$ for $1 \leq j < d$ and $x_d = 0$. If successful, set $y_1 \ldots y_n \leftarrow x_1 \ldots x_n$. Return to F2. ∎

Even better is to incorporate a similar procedure into the solver itself; see exercise 275.

Figure 23.3: The solution

Chapter 24

Glossary (SAT)

- clause - disjunction of one or more literals. For example: var1 OR -var2 OR var3 ... - at least one literal must be satisfied in each clause.
- CNF (conjunctive normal form) formula, conjunction of one or more clauses. Is a list of clauses, all of which must be satisfied.
- literal/term - can be variable and -variable, these are different literals.
- pure literal - present only as x or -x. Can be eliminated at start.
- unit clause - clause with only one literal.
- DIMACS - file format, popular among SAT solvers, benchmarks, etc. Earliest description (1993): http://www.domagoj-babic.com/uploads/ResearchProjects/Spear/dimacs-cnf.pdf. ”DIMACS” acronym itself means ”The DIMACS Implementation Challenges”[1].

[1] http://archive.dimacs.rutgers.edu/Challenges/

Chapter 25

Glossary (SMT)

- sort - datatype.

Chapter 26

Further reading

- Julien Vanegue, Sean Heelan, Rolf Rolles – SMT Solvers for Software Security [1]
- Armin Biere, Marijn Heule, Hans van Maaren, Toby Walsh – Handbook of Satisfiability (2009). The second edition came out in 2021.
- Rui Reis – Practical Symbolic Execution and SATisfiability Module Theories (SMT) 101 [2].
- Daniel Kroening and Ofer Strichman – Decision Procedures – An Algorithmic Point of View [3].

26.1 Z3-specific

- Z3 API in Python [4]
- Z3 Strategies [5]
- Nikolaj Bjørner – Recent Trends in SMT and Z3: An interactive taste of SMT with Z3 [6].
- Nikolaj Bjørner, Leonardo de Moura, Lev Nachmanson, Christoph Wintersteiger – Programming Z3 [7].
- Questions tagged [z3] on Stack Overflow [8].

26.2 SAT-specific

Instead of epigraph: "There is a 350+ book from 2015 by Donald Knuth in the art of programming series on SAT. So the nerdiness of SAT has highest blessing."[9].

- Donald Knuth – TAOCP 7.2.2.2. Satisfiability [10]. Since the Postscript file is freely available at Donald Knuth's website, I converted it to PDF and put it to my website: Download here.
- Niklas Een, Niklas Sorensson – Translating Pseudo-Boolean Constraints into SAT [11].
- Armin Biere – Using High Performance SAT and QBF Solvers[12].
- Armin Biere – Tutorial on Model Checking (2008)[13] – short history of SAT solvers and model checking.
- https://en.wikipedia.org/wiki/Tseytin_transformation. G. S. Tseitin – On the Complexity of Derivation in Propositional Calculus[14].

[1] https://yurichev.com/mirrors/SMT/woot12.pdf

[2] http://deniable.org/reversing/symbolic-execution

[3] http://www.decision-procedures.org

[4] http://www.cs.tau.ac.il/~msagiv/courses/asv/z3py/guide-examples.htm

[5] http://www.cs.tau.ac.il/~msagiv/courses/asv/z3py/strategies-examples.htm

[6] http://www.cse.chalmers.se/~laurako/links/ADuctSlides/L10.html

[7] http://theory.stanford.edu/~nikolaj/programmingz3.html

[8] https://stackoverflow.com/questions/tagged/z3

[9] https://twitter.com/ArminBiere/status/1288132283443142661

[10] http://www-cs-faculty.stanford.edu/~knuth/fasc6a.ps.gz

[11] http://minisat.se/downloads/MiniSat+.pdf

[12] http://fmv.jku.at/biere/talks/Biere-TPTPA11.pdf

[13] http://fmv.jku.at/biere/talks/Biere-AB08-talk.pdf

[14] https://yurichev.com/mirrors/SAT/Tseitin/

- Martin Finke – Equisatisfiable SAT Encodings of Arithmetical Operations [15].
- Satisfiability / Suggested Format (1993) [16].

26.3 SMT-specific

- Important website: http://smtlib.cs.uiowa.edu/.
- Theories: http://smtlib.cs.uiowa.edu/logics.shtml, https://cs.nyu.edu/pipermail/smt-lib/2005/000046.html.
- SMT-COMP mailing list: https://cs.nyu.edu/mailman/listinfo/smt-comp.
- SMT-LIB mailing list: https://cs.nyu.edu/pipermail/smt-lib/, https://groups.google.com/forum/#!forum/smt-lib.
- SMT-LIB standard: http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2017-07-18.pdf.
- David R. Cok – The SMT-LIBv2 Language and Tools: A Tutorial: http://smtlib.github.io/jSMTLIB/SMTLIBTutorial.pdf.

26.3.1 Benchmarks

SAT/SMT is a young field (yet). Not much documentation exist.

So I've found SMT benchmark collection as an invaluable source of information. Benchmarks are typically generated by a SMT applications, so you can get into better understanding, how SMT is used.

Using gitlab at uiowa.edu[17]:

```
sudo apt-get install git-lfs
git clone https://clc-gitlab.cs.uiowa.edu:2443/mpreiner/smt-lib-non-incremental.git
cd smt-lib-non-incremental
git submodule update --init
cd ..
git clone https://clc-gitlab.cs.uiowa.edu:2443/mpreiner/smt-lib-incremental.git
cd smt-lib-incremental
git submodule update --init
cd ..
```

Using starexec[18], pick this:

- "do you want to download the single space or the hierarchy?" — hierarchy
- "do you want to download the benchmarks and/or the solvers?" — solvers + benchmarks
- "do you want to store benchmarks/solvers in id directories?" — no

These benchmarks are used in SMT-COMP competition[19].

26.4 et cetera

Many puzzles from Martin Gardner's books can be solved effortlessly using SAT/SMT. And it is such a fun, because SAT/SMT solvers can help understanding all the math theory behind them.

Also recommended by Armin Biere:

- Stuart Russell and Peter Norvig – Artificial Intelligence: A Modern Approach
- Helmut Veith, Edmund M. Clarke, Thomas A. Henzinger – Handbook of Model Checking
- Christos Papadimitriou – Computational Complexity (1994)

[15] http://www.martin-finke.de/documents/Masterarbeit_bitblast_Finke.pdf
[16] https://yurichev.com/mirrors/SAT/dimacs-cnf.pdf
[17] https://clc-gitlab.cs.uiowa.edu:2443/explore/groups
[18] https://www.starexec.org/starexec/secure/explore/spaces.jsp?id=1
[19] https://smt-comp.github.io/

26.5 Henry Warren – Hacker's Delight

Some people say these branchless tricks and hacks were only relevant for old RISC CPUs, so you don't need to read it. Nevertheless, these hacks and understanding them helps immensely to get into boolean algebra and all the mathematics.

Also, such hacks got their second life to be used in cryptography. Many important primitives should be executed in constant time, to prevent side-channel attacks.

Listing 26.1: A piece of code from OpenSSH 7.x

```
/*
 * increment counter 'ctr',
 * the counter is of size 'len' bytes and stored in network-byte-order.
 * (LSB at ctr[len-1], MSB at ctr[0])
 */
static inline void
aesctr_inc(u8 *ctr, u32 len)
{
        ssize_t i;

#ifndef CONSTANT_TIME_INCREMENT
        for (i = len - 1; i >= 0; i--)
                if (++ctr[i])   /* continue on overflow */
                        return;
#else
        u8 x, add = 1;

        for (i = len - 1; i >= 0; i--) {
                ctr[i] += add;
                /* constant time for: x = ctr[i] ? 1 : 0 */
                x = ctr[i];
                x = (x | (x >> 4)) & 0xf;
                x = (x | (x >> 2)) & 0x3;
                x = (x | (x >> 1)) & 0x1;
                add *= (x^1);
        }
#endif
}
```

Chapter 27

Some applications

- All sorts of theorem provers, including (but not limited to) Isabelle[1], HOL...
- Dafny (Microsoft Research)[2], uses Z3.
- KLEE[3] (uses STP).
- CBMC – Bounded Model Checker for C and C++ programs: http://www.cprover.org/cbmc/. Short intro to its innards: http://www.cprover.org/cbmc/doc/cbmc-slides.pdf.
- Firewall checker, can check equivalence of two firewalls. https://github.com/Z3Prover/FirewallChecker. Nikolaj Bjørner and Karthick Jayaraman – Checking Cloud Contracts in Microsoft Azure [4], Karthick Jayaraman, Nikolaj Bjørner, Geoff Outhred, Charlie Kaufman – Automated Analysis and Debugging of Network Connectivity Policies [5].
- Frama-C – a static analyzer, inspects programs without executing them [6].
- VCC: A Verifier for Concurrent C [7] – a competitor to Frama-C from Microsoft Research. Seems to be stalled.
- Boogie: An Intermediate Verification Language [8] (Microsoft Research). Used as a bridge between VCC and Z3.
- Spec# (Microsoft Research)
- Why3 – a platform for deductive program verification, used in Frama-C[9].
- RISC-V Formal Verification Framework[10]
- Yosys – a framework for Verilog RTL synthesis[11].
- IVy – a tool for specifying, modeling, implementing and verifying protocols[12]. Uses Z3.
- Cryptol [13]: a language for cryptoalgorithms specification and proving it's correctness. Uses Z3.
- SPARK – a formally defined computer programming language based on the Ada. It can use Alt-Ergo, Z3, CVC4, etc.
- LiquidHaskell [14].
- Google's Operations Research tools has SAT/MaxSAT solver as an engine [15].

[1] http://isabelle.in.tum.de/

[2] https://en.wikipedia.org/wiki/Dafny

[3] https://klee.github.io/

[4] https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/nbjorner-icdcit2015.pdf

[5] https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/secguru.pdf

[6] http://frama-c.com/

[7] https://www.microsoft.com/en-us/research/project/vcc-a-verifier-for-concurrent-c/

[8] https://www.microsoft.com/en-us/research/project/boogie-an-intermediate-verification-language/

[9] http://why3.lri.fr/

[10] https://github.com/SymbioticEDA/riscv-formal

[11] http://www.clifford.at/yosys/

[12] http://microsoft.github.io/ivy/

[13] http://cryptol.net, https://github.com/GaloisInc/cryptol

[14] https://ucsd-progsys.github.io/liquidhaskell-blog/

[15] https://github.com/google/or-tools/tree/v7.0/ortools/sat

- Musketeer – A static analysis approach to automatic [memory] fence insertion [16].
- Averest is a framework for the specification, verification, and implementation of reactive systems [17].
- OpenJML – a program verification tool for Java programs that allows you to check the specifications of programs annotated in the Java Modeling Language [18]. See also: ESC/Java.
- Souper: A Synthesizing Superoptimizer[19]
- See the list of ”Use cases” [20] where the STP solver is used. Also, for CVC4[21].
- Linux Kernel folks got interested in SAT: https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html#full-sat-solver-for-kconfig, https://kernelnewbies.org/KernelProjects/kconfig-sat.

27.1 Compiler's optimization verification

Your compiler optimized something out, but you're unsure if your optimization rules are correct, because there are lots of them. You can prove the original expression and the optimized are equal to each other.

This is what I did for my toy decompiler: 16.6.2.

“Alive” project:

Nuno P. Lopes, David Menendez, Santosh Nagarakatte, John Regehr – Practical Verification of Peephole Optimizations with Alive [22].

Another paper: Provably Correct Peephole Optimizations with Alive [23].

At github: https://github.com/nunoplopes/alive.

Nuno Lopes – Verifying Optimizations using SMT Solvers [24].

27.2 All sorts of papers and articles I've found interesting

- Martin Hořeňovský about master-key system [25].

[16] https://arxiv.org/pdf/1312.1411.pdf, http://www.cprover.org/wmm/musketeer/

[17] http://www.averest.org/

[18] http://www.openjml.org/

[19] https://arxiv.org/abs/1711.04422, https://github.com/google/souper, https://news.ycombinator.com/item?id=10463312, https://blog.regehr.org/archives/1252.

[20] https://github.com/stp/stp/blob/master/docs/index.rst#use-cases

[21] https://cvc4.github.io/third-party-applications.html

[22] http://web.ist.utl.pt/nuno.lopes/pubs/alive-cacm18.pdf

[23] http://www.cs.utah.edu/~regehr/papers/pldi15.pdf

[24] https://llvm.org/devmtg/2013-11/slides/Lopes-SMT.pdf

[25] https://codingnest.com/modern-sat-solvers-fast-neat-and-underused-part-2-of-n/, https://codingnest.com/files/thesis.pdf

Chapter 28

Acronyms used

GCD	Greatest Common Divisor	7
LCM	Least Common Multiple	7
CNF	Conjunctive normal form	11
WCNF	Weighted Conjunctive normal form	391
DNF	Disjunctive normal form	27
DSL	Domain-specific language	15
CPRNG	Cryptographically Secure Pseudorandom Number Generator	75
SMT	Satisfiability modulo theories	3
SAT	Boolean satisfiability problem	3
LCG	Linear congruential generator	4
PL	Programming Language	13
OOP	Object-oriented programming	472
SSA	Static single assignment form	17
CPU	Central processing unit	174
FPU	Floating-point unit	495
PRNG	Pseudorandom number generator	4
CRT	C runtime library	506
CRC	Cyclic redundancy check	368

AST	Abstract syntax tree	474
AKA	Also Known As	6
CTF	Capture the Flag	550
ISA	Instruction Set Architecture	442
CSP	Constraint satisfaction problem	239
CS	Computer science	13
DAG	Directed acyclic graph	97
NOP	No Operation	446
JVM	Java Virtual Machine	495
VM	Virtual Machine	509
LZSS	Lempel–Ziv–Storer–Szymanski	386
RAM	Random-access memory	174
FPGA	Field-programmable gate array	174
EDA	Electronic design automation	552
MAC	Message authentication code	554
ECC	Elliptic curve cryptography	554
API	Application programming interface	15
NSA	National Security Agency	75
DPLL	Davis–Putnam–Logemann–Loveland	642
RE	Regular Expression	163
RSA	Rivest–Shamir–Adleman cryptosystem	4
ITE	If-Then-Else	5
DFA	Deterministic finite automaton	5

TAOCP	The Art Of Computer Programming (Donald Knuth's book)	69
DES	Data Encryption Standard	552
ALU	Arithmetic logic unit	87
EE	Electrical engineering	270
PCB	Printed circuit board	6
ETA	Estimated time of arrival	34
CDCL	Conflict-driven clause learning	13
MSB	Most Significant Bit	138
ILP	Integer Linear Programming	43
IFF	If and only if: $\iff$	131
UF	Uninterpreted Function	82
MUS	Minimal Unsatisfiable Subformula	597
MCS	Minimal Correction Subset	597
OEIS	On-Line Encyclopedia of Integer Sequences	359
CSV	Comma-Separated Values	199
ROM	Read-Only Memory	174
ASIC	Application-specific integrated circuit	174
GPU	Graphics Processing Unit	174
NFA	Nondeterministic Finite Automaton	179
FA	Finite Automaton	162
BTW	By The Way	179
JSON	JavaScript Object Notation	179
LS	Latin Square	357
MOLS	Mutually Orthogonal Latin Square	7