PhD Thesis

Algorithms and Lower Bounds for All-Pairs Max-Flow

Thesis for the Ph.D. Degree

by

Ohad Trabelsi

מכון ויצמן למדע
WEIZMANN INSTITUTE OF SCIENCE

Supervisors: Prof. Robert Krauthgamer and Prof. Eden Chlamtác
Department of Computer Science and Applied Mathematics
The Weizmann Institute of Science

Submitted to Feinberg Graduate School
Weizmann Institute of Science
Rehovot, Israel

September 16, 2020

(Revised November 2020)

Abstract

Ever since its introduction in the 1950s, the maximum flow problem has been extensively studied and became a central algorithmic tool with numerous applications. This problem, denoted Max-Flow, asks to ship the largest possible amount of flow from a source node to a sink node in a given edge-capacitated graph. Shortly after Max-Flow was introduced, Gomory and Hu (1961) investigated the problem of computing the Max-Flow values between all pairs of nodes in the same graph, called All-Pairs Max-Flow. Their seminal work established that in the undirected setting All-Pairs Max-Flow can be solved using only $n-1$ executions of Max-Flow, where n denotes the number of nodes in the graph and m the number of edges; moreover, these Max-Flow values and their corresponding minimum cuts could be stored in a succinct data structure consisting of a tree on the same set of nodes as the input graph, nowadays called a Gomory-Hu tree. Despite a lot of research aimed at extending this result to directed graphs, it is still not known how to solve Max-Flow for all pairs faster than naively solving it separately for each pair.

Our first set of results tackles this gap by showing that well-known time-complexity hypotheses must be broken in order to solve All-Pairs Max-Flow faster. We show for the directed setting a conditional lower bound based on the 3OV hypothesis, which asserts that deciding if n short $0-1$ vectors contain three vectors that are orthogonal to each other requires time $n^{3-o(1)}$, and consequently on the Strong Exponential Time Hypothesis (SETH) which is perhaps the most popular time-complexity hypothesis in the field. This bound is strongest for sparse graphs, and we extend it to dense graphs by relying on another hypothesis regarding the time it takes to decide if an input graph contains a 4-clique. Additionally, we extend the 3OV-hardness to undirected graphs with node capacities.

Our second set of results consists of new algorithms for All-Pairs Max-Flow. After the breakthrough by Gomory and Hu, undirected graphs have resisted progress for many decades, and all faster algorithms were by-products of faster algorithms for Max-Flow. This was recently challenged with the introduction of a tree-packing approach by Bhalgat et al. (2007) for the unit-capacity setting, resulting in an algorithm for Gomory-Hu trees with running time $\tilde{O}(mn)$, which can also be obtained by a more careful analysis of Karger and Levine's (2002) algorithm for Max-Flow.

This $\tilde{O}(mn)$ running time is the best one could hope for using the half-century old Gomory-Hu method. We overcome this obstacle by devising a new algorithm for Gomory-Hu trees in unit-capacity graphs, which combines the Gomory-Hu and the tree-packing methods; it runs in time $\tilde{O}(m^{3/2}n^{1/6})$, which is faster than $\tilde{O}(mn)$ whenever $m \leq \tilde{O}(n^{5/3})$, and with faster Max-Flow algorithms it would be faster for all densities. We also show that unlike the directed case, SETH is unlikely to preclude even a near-linear time $\tilde{O}(m)$ algorithm for this problem! This is achieved by designing a nondeterministic $\tilde{O}(m)$ time algorithm for Gomory-Hu trees in this setting, and applying the framework of Carmasino et al. (2016).

In our third and final set of results we consider the All-Pairs Max-Flow problem from the perspective of data structures. Perhaps surprisingly, we show that data structures based on Gomory-Hu trees are essentially optimal, in the sense that any data structure for minimum-cut queries with near-linear preprocessing time and polylogarithmic (amortized) query time can be used to construct a Gomory-Hu tree in near-linear time (which is still open), even if the queries are restricted to a fixed source. The techniques used for this allows us to design also new algorithms for the approximation case.

This thesis is based on the following papers.

1. Robert Krauthgamer and Ohad Trabelsi. Conditional Lower Bounds for All-Pairs Max-Flow. *ACM Transactions on Algorithms*, 14(4):42:1–42:15, August 2018. Preliminary version in *ICALP 2017*.

2. Amir Abboud, Loukas Georgiadis, Giuseppe F. Italiano, Robert Krauthgamer, Nikos Parotsidis, Ohad Trabelsi, Przemyslaw Uznanski, and Daniel Wolleb-Graf. Faster Algorithms for All-Pairs Bounded Min-Cuts. In *46th International Colloquium on Automata, Languages, and Programming*, pages 7:1–7:15, July 2019.

3. Amir Abboud, Robert Krauthgamer, and Ohad Trabelsi. New Algorithms and Lower Bounds for All-Pairs Max-Flow in Undirected Graphs. In *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms*, pages 48-61, January 2020. Accepted to *Theory of Computing.*

4. Amir Abboud, Robert Krauthgamer, and Ohad Trabelsi. Cut-Equivalent Trees are Optimal for Min-Cut Queries. Accepted to *FOCS 2020.*

The following papers were written during my PhD period but are not a part of the thesis, and only briefly discussed, in appendix A.

- Robert Krauthgamer and Ohad Trabelsi, ”The Set Cover Conjecture and Subgraph Isomorphism with a Tree Pattern”. In *36th International Symposium on Theoretical Aspects of Computer Science 2019*, pages 45:1–45:15, March 2019.

- Arnold Filtser, Robert Krauthgamer, and Ohad Trabelsi, ”Relaxed Voronoi: A Simple Framework for Terminal-Clustering Problems”. In *Proceedings of the 2nd Symposium on Simplicity in Algorithms*, pages 10:1–10:14, January 2019.

- Lior Kamma and Ohad Trabelsi, ”Nearly Optimal Time Bounds for kPath in Hypergraphs”. Manuscript, August 2018.

Declaration. I, Ohad Trabelsi, declare that this thesis summarizes my independent work under the supervision of Professor Robert Krauthgamer, where Sections 3, 4, and 5 were obtained together also with Amir Abboud. In addition, only my own contribution to paper number 2 is included (this paper is the result of a merge of two papers).

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my advisor, Robert Krauthgamer, for guiding me in my own path, and always keeping his door open for me. I am very grateful for the countless fascinating discussions, smart advice, continuous support, and inspiring patience. I learned a lot from him, and for that I am indebted. Additionally, I am incredibly lucky to have Amir Abboud as an amazing mentor, an insightful coauthor, and a great friend. I thank him for collaborating with me and giving me advice, and for hosting me for a wonderful summer internship at IBM Almaden. I also wish to thank my second advisor Eden Chlamtac for the support throughout the PhD period.

I happily acknowledge the Weizmann Institute theory group for such an amazing academic environment through the seminars and the good spirits. Also, I thank the members of Robi's group throughout the years for the discussions and the laughs.

Special gratitude goes to my family, and specifically to my parents Eliyahu and Shula, for their unconditional love and support throughout this journey.

"The most beautiful thing we can experience is the mysterious"

Albert Einstein

Contents

1 Introduction **1**

1.1 Conditional Lower Bounds for the Directed Case . . . 2

1.2 Improved Lower Bounds for Unit-Capacity Directed Graphs . . . 3

1.3 New Results for Undirected Graphs . . . 3

1.4 Optimality of Gomory-Hu Trees for Min-Cut Queries and Fast Approximation Algorithms . . . 4

2 Conditional Lower Bounds for All-Pairs Max-Flow in Directed Graphs **5**

2.1 Introduction . . . 5

2.1.1 Prior Work . . . 6

2.1.2 Our Contribution . . . 7

2.2 Reduction to Multiple-Pairs Max-Flow with Unit Capacity . . . 10

2.3 Reduction to Multiple-Pairs Max-Flow in Capacitated Networks . . . 13

2.4 Generalization to Bounded Cuts . . . 16

2.5 Global Max-Flow . . . 17

3 Faster Algorithms for All-Pairs Bounded Min-Cuts in Unit-Capacity Directed Graphs **19**

3.1 Introduction . . . 19

3.1.1 Our Contribution . . . 21

3.2 Preliminaries . . . 22

3.3 Overview of Our Algorithmic Approach . . . 23

3.4 Reducing 4-Clique to All-Pairs Min-Cut . . . 23

3.4.1 Reduction to the Unbounded Case . . . 24

3.4.2 Reduction to the k-Bounded Case . . . 26

3.5 Randomized Algorithms for General Digraphs . . . 28

4 New Algorithms and Lower Bounds for All-Pairs Max-Flow in Unit-Capacity Undirected Graphs **31**

4.1 Introduction . . . 31

4.1.1 The Challenge of Lower Bounds in Undirected Graphs . . . 33

4.1.2 Our Results . . . 34

4.2 Algorithm for a Cut-Equivalent Tree . . . 36

4.3 Near-Linear Nondeterministic Algorithm for Cut-Equivalent Tree . . . 38

4.3.1 The Nondeterministic Algorithm . . . 39

4.3.2 Reduction from a Decision Problem to a total search problem . . . 45

4.4 Conditional Lower Bound for All-Pairs Max-Flow 46
4.5 Open Problems 50
4.6 Acknowledgements 51

5 Cut-Equivalent Trees are Optimal for Min-Cut Queries 53
5.1 Introduction 53
5.1.1 Our Results 55
5.1.2 Preliminaries 58
5.2 Our Approximation Algorithms 58
5.2.1 Overview 58
5.2.2 Approximate Min-Cut Queries and Flow-Equivalent Trees 62
5.2.3 Our Tree-Like Data Structure 63
5.2.4 A Faster Implementation For Unweighted Graphs 70
5.2.5 Handling Randomized Data Structures 72
5.3 Algorithm for a Cut-Equivalent Tree 73
5.3.1 The Algorithm for General Capacities 74
5.3.2 Overview of the Algorithm 74
5.3.3 Full Algorithm 75
5.3.4 Analysis 76
5.3.5 Lifting the Assumption on Max-Flow Queries 82
5.3.6 Unique Cut-Equivalent Tree via Pertubation 83
5.4 Algorithm for an Output Sensitive Data Structure 84
5.5 Algorithm for Flow-Equivalent Trees 86
5.5.1 Recovering Ultrametrics 88

A Papers not Included in the Thesis 97

Chapter 1

Introduction

The design of efficient algorithms has always been an intriguing and thriving area of research, and many fundamental problems were successfully studied, for example maximum flow in a network and matrix multiplication. However, the progress in designing fast algorithms for many core problems has halted for a very long time. Hence, ideally we would seek a proof that a problem with running time $T(n)$ that has not been improved for decades, actually requires time $T(n)^{1-o(1)}$, explaining the lack of progress. Unfortunately, such unconditional lower bounds seem unlikely in most cases. For example, not even super-linear lower bounds are known for SAT. Since classical NP-hardness techniques seem too coarse to show tight conditional lower bounds, a new, more fine-grained approach was looked for. A recent line of work that was initiated in order to address this hurdle, mimicks NP-hardness style lower bounds in the following sense. First, select a few key problems that are conjectured to require $T(n)^{1-o(1)}$ running time, and then use fine-grained reductions to derive hardness for other problems. Our work further advances this area of research, in particular for the problem of computing the maximum flow value between a given source-sink pair s, t, denoted Max-Flow, between multiple pairs of nodes; when can it be done faster than computing Max-Flow separately for each pair? when it can not?

Max-Flow is a fundamental problem in computer science. This classical problem and its variations were studied extensively over the past decades and have become key algorithmic tools with numerous applications, in theory and in practice. Moreover, techniques developed for flow problems were generalized or adapted to other problems, see for example [BJS10, AMO93, AHK12]. A summary of state-of-the-art Max-Flow algorithms follows, for both the directed and the undirected versions. Throughout, n denotes the number of nodes in a graph, m the number of edges, and U the maximum value of capacities, assumed to be integers (although, we focus on polynomially-bounded capacities). For general capacities, an algorithm by Lee and Sidford [LS14] has running time $\tilde{O}(m\sqrt{n}\log U)$, while for small capacities, faster algorithms exist [Mad16, LS20a, LS20b] that run in time $O(\min\{m^{10/7}U^{1/7}, m^{11/8}U^{1/4}, m^{4/3}U^{1/3}\})$. If in addition the graph is undirected then an algorithm by Karger and Levine [KL15] could be applied with running time $\tilde{O}(m + nv)$, where v is the maximum flow value.

A very natural problem is to compute the maximum st-flow for multiple source-sink pairs in the same graph G, where the problem of reporting Max-Flow for all $\binom{n}{2}$ source-sink pairs is called All-Pairs Max-Flow. This multi-terminal problem, dating back to 1960 [May60, Chi60], is the main focus of this thesis. The seminal work of Gomory and Hu [GH61] shows that in undirected graphs, All-Pairs Max-Flow requires at most $n-1$ executions of Max-Flow (see also [Gus90], where the $n-1$ computations are all on the input graph), and a lot of research aimed to extend this result to directed graphs, with several partial successes. However, for general directed graphs it is still not known how to solve Max-Flow for multiple source-sink pairs faster than solving it separately for each pair. In this thesis we consider both the directed and the undirected cases, providing exciting new lower bounds (conditional hardness) and upper bounds (algorithms) for All-Pairs Max-Flow.

1.1 Conditional Lower Bounds for the Directed Case

The first part of the thesis presents our new conditional lower bounds for All-Pairs Max-Flow in directed graphs. Specifically, we provide evidence that computing the maximum flow value between every pair of nodes in a *directed* graph with capacity bounded by $U \leq n$ cannot be solved in time that is significantly faster (i.e., by a polynomial factor) than $O(n^3)$ even for sparse graphs, namely $m = O(n)$; in particular, it rules out an algorithm that runs (for all m) significantly faster than $O(n^2 m)$. Since a single maximum st-flow can be solved in time $\tilde{O}(m\sqrt{n})$ [LS14], we conclude that conditionally, the all-pairs version requires time equivalent to $\tilde{\Omega}(n^{3/2})$ computations of maximum st-flow, which strongly separates the directed case from the undirected one. Moreover, if maximum st-flow can be solved in time $\tilde{O}(m)$, then the time of $\tilde{\Omega}(n^2)$ computations is needed. This is in contrast to a conjecture of Lacki, Nussbaum, Sankowski, and Wulff-Nilsen [LNSW12] that All-Pairs Max-Flow in general graphs can be solved faster than the time of $O(n^2)$ computations of maximum st-flow.

Specifically, we show that in sparse graphs $G = (V, E, w)$, if one can compute the maximum st-flow from every s in an input set of sources $S \subseteq V$ to every t in an input set of sinks $T \subseteq V$ in time $O((|S||T|m)^{1-\varepsilon})$, for some $|S|$, $|T|$ and a constant $\varepsilon > 0$, then a problem known as MAX-CNF-SAT (maximum satisfiability of conjunctive normal form formulas) with n' variables and m' clauses can be solved in time ${m'}^{O(1)} 2^{(1-\delta)n'}$ for a constant $\delta(\varepsilon) > 0$, a problem for which not even $2^{n'}/\operatorname{poly}(n')$ algorithms are known[1]. Such running time for MAX-CNF-SAT would in particular refute the Strong Exponential Time Hypothesis, abbreviated SETH (see [IP01, Vas18]). Hence, we improve the lower bound of Abboud, Vassilevska-Williams, and Yu [AWY18], who showed that for every fixed $\varepsilon > 0$ and $|S| = |T| = O(\sqrt{n})$, if the above problem can be solved in time $O(n^{3/2-\varepsilon})$, then some incomparable (and intuitively weaker) conjecture is false. Furthermore, a larger lower bound than ours implies that the maximum st-flow problem requires strictly super-linear time, which would be an amazing breakthrough.

In addition, we show that All-Pairs Max-Flow in *unit*-capacity networks with every edge-density $m = m(n)$, cannot be computed in time significantly faster than $O(mn)$, even for acyclic networks. The gap to the fastest known algorithm by Cheung, Lau, and Leung [CLL13] is a factor of $O(m^{\omega-1}/n)$, and for directed acyclic graphs (DAGs) it is $O(n^{\omega-2})$, where $\omega < 2.38$ is the matrix multiplication exponent. Finally, we extend our lower bounds to the version that asks only for the maximum-flow values below a given threshold (over all source-sink pairs).

These results are presented in Section 2, and essentially replicate our published paper [KT18b].

[1] It was later shown [ABDN18] that MAX-CNF-SAT can be reduced to 3OV. As our results hold also with 3OV as the hardness hypothesis, 3OV should be seen as our hardness hypothesis instead.

1.2 Improved Lower Bounds for Unit-Capacity Directed Graphs

The conditional lower bounds for the unit-capacity setting presented in Section 1.1 leave a big gap to the fastest known algorithms, and so we focus here on this setting. Specifically, for *directed graphs* with *unit* edge/vertex capacities (hence Max-Flow corresponds to edge/vertex connectivity), we deal with the *k-bounded* case, where the algorithm has to find all pairs with Max-Flow value less than k, and report only those. The most basic case $k = 1$ is the Transitive Closure (TC) problem, which can be solved in time $O(mn)$ combinatorially (i.e. not using fast matrix multiplication techniques), and in time $O(n^\omega)$ generally, where $\omega < 2.38$ is the matrix-multiplication exponent. These time bounds are conjectured to be optimal.

We present new algorithms and conditional lower bounds that advance the frontier for $k \in [n]$, as follows:

- The first super-cubic lower bound of $n^{\omega-1-o(1)}k^2$ time (which is meaningful for $k \geq \Omega(\sqrt{n})$) under the 4-Clique conjecture, which holds even in the simplest case of DAGs with unit vertex-capacities. It improves on the previous (SETH-based) lower bounds for unit edge-capacities described in Section 1.1, even in the unbounded setting $k = n$. For combinatorial algorithms, our reduction implies an $n^{2-o(1)}k^2$ conditional lower bound. Thus, we identify new settings where the complexity of the problem is (conditionally) higher than that of TC.

- An algorithm for unit *vertex-capacities* that runs in time $O((nk)^\omega)$. This is only a factor k^ω away from the bound for TC, and nearly matches it for all $k = n^{o(1)}$.

Our first result arises from a novel reduction of a different structure than the SETH-based constructions, while the second one adapts the network coding method of Cheung, Lau, and Leung [CLL13] to vertex-capacitated digraphs. These results are presented in Section 3, and essentially replicate the corresponding parts in our published paper [AGI$^+$19].

1.3 New Results for Undirected Graphs

If Max-Flow can be solved in time $T(m)$, then an $O(n^2) \cdot T(m)$ is a trivial upper bound for All-Pairs Max-Flow. But can we do better? For directed graphs, this time bound might be optimal (e.g. see Section 1.1). In contrast, for undirected graphs with edge capacities, the seminal algorithm of Gomory and Hu [GH61] runs in a much faster time $O(n) \cdot T(m)$, and under the plausible assumption that Max-Flow can be solved in near-linear time $\tilde{O}(m)$, this half-century old algorithm yields an $\tilde{O}(mn)$ bound. Gomory and Hu's algorithm additionally constructs a cut-equivalent tree, which is a tree on the same set of nodes as the input graph such that every minimum st-cut in the tree is a minimum st-cut in the input graph. Several other algorithms have been designed through the years for cut-equivalent tree construction, including $\tilde{O}(mn)$ time for unit-capacity edges (unconditionally), but none of them break the $O(mn)$ barrier. Meanwhile, no super-linear lower bound was shown for undirected graphs.

In the third part of the thesis, we present three different results about the time-complexity of All-Pairs Max-Flow in undirected graphs.

- For *node capacities*, we design the first conditional lower bounds for All-Pairs Max-Flow, giving an essentially optimal lower bound. Our lower bound is $n^{3-o(1)}$ conditioned on the 3OV conjecture, which asserts that finding three vectors that are orthogonal to each other among a set of n vectors in $\{0,1\}^d$, for $d \geq \omega(\log n)$, requires $n^{3-o(1)}$ time, and on SETH as a consequence.

- For edge capacities, our efforts to prove similar lower bounds have failed, but we have discovered a surprising new algorithm that breaks the $O(mn)$ barrier for graphs with unit-capacity edges! Assuming $T(m) = m^{1+o(1)}$, our algorithm runs in time $m^{3/2+o(1)}$ and outputs a cut-equivalent tree (similarly to the Gomory-Hu algorithm). Even with current Max-Flow algorithms we improve state-of-the-art as long as $m = O(n^{5/3-\varepsilon})$.

- Finally, we explain the lack of lower bounds by proving a *non-reducibility* result (i.e., that a reduction from SETH to constructing Gomory-Hu trees is unlikely). This result is based on a new near-linear time $\tilde{O}(m)$ *non-deterministic* algorithm for constructing a cut-equivalent tree which may be of independent interest.

These results are presented in Section 4 and essentially replicate our published paper [AKT20b].

1.4 Optimality of Gomory-Hu Trees for Min-Cut Queries and Fast Approximation Algorithms

Finally, we analyze the connection between data structures for minimum cut (Min-Cut) queries and cut-equivalent trees for the undirected, edge-capacities setting. In a Min-Cut data structure the input graph is preprocessed to quickly report a minimum-capacity cut that separates a query pair of nodes s, t. The best data structure known for this problem simply builds a *cut-equivalent tree*, discovered 60 years ago by Gomory and Hu [GH61], who also showed how to construct it using $n-1$ minimum *st*-cut computations. Using state-of-the-art algorithms for minimum *st*-cut [LS14], one can construct the tree in time $\tilde{O}(mn^{3/2})$, which is also the preprocessing time of the data structure.

Our main result shows the following equivalence: Cut-equivalent trees can be constructed in near-linear time if and only if there is a data structure for Min-Cut queries with near-linear preprocessing time and polylogarithmic (amortized) query time, and even if the queries are restricted to a fixed source. That is,cut-equivalent trees are an essentially optimal solution for Min-Cut queries. This equivalence holds even for every minor-closed family of graphs, such as bounded-treewidth graphs, for which a two-decade old data structure [ACZ98] implies the first near-linear time construction of cut-equivalent trees.

Moreover, unlike all previous techniques for constructing cut-equivalent trees, ours is robust to relying on *approximation algorithms*. In particular, using the almost-linear time algorithm for $(1+\epsilon)$-approximate minimum *st*-cut [KLOS14] we can construct a $(1+\epsilon)$-approximate flow-equivalent tree (which is a slightly weaker notion) in time $n^{2+o(1)}$. This leads to the first $(1+\epsilon)$-approximation for All-Pairs Max-Flow that runs in time $n^{2+o(1)}$, and matches the output size almost-optimally. These results are presented in Section 5 and essentially replicate our published paper [AKT20a].

Chapter 2

Conditional Lower Bounds for All-Pairs Max-Flow in Directed Graphs[1]

2.1 Introduction

The maximum flow problem is one of the most fundamental problems in combinatorial optimization. This classic problem and its variations such as minimum-cost flow, integral flow, and minimum-cost circulation, were studied extensively over the past decades, and have become key algorithmic tools with numerous applications, in theory and in practice. Moreover, techniques developed for flow problems were generalized or adapted to other problems, see for example [BJS10, AMO93, AHK12]. The maximum st-flow problem, which we shall denote Max-Flow, asks to ship the maximum amount of flow from a source node s to a sink node t in a directed edge-capacitated graph $G = (V, E, w)$, where throughout this chapter, we denote $n = |V|$ and $m = |E|$, and assume integer capacities bounded by U. After this problem was introduced in 1954 by Harris and Ross (see [Sch02] for a historical account), Ford and Fulkerson [FF56] devised the first algorithm for Max-Flow, which runs in time $O((n+m)F)$, where F is the maximum value of a feasible flow. Ever since, a long line of generalizations and improvements was studied, and the current fastest algorithm for Max-Flow with arbitrary capacities is by Lee and Sidford [LS14], which takes $O(m\sqrt{n}\log U)$ time. For the case of small capacities and sufficiently sparse graphs, faster algorithms are known [Mad16, LS20a, LS20b], that run in time $\tilde{O}(\min\{m^{10/7}U^{1/7}, m^{11/8}U^{1/4}, m^{4/3}U^{1/3})$. Here and throughout this chapter, $\tilde{O}(f)$ denotes $O(f\log^c f)$ for unspecified constant $c > 0$.

A very natural problem is to compute the maximum st-flow for multiple source-sink pairs in the same graph G. The seminal work of Gomory and Hu [GH61] shows that in undirected graphs, Max-Flow for all $\binom{n}{2}$ source-sink pairs requires at most $n-1$ executions of Max-Flow (see also [Gus90], where the $n-1$ computations are all on the input graph), and a lot of research aimed to extend this result to directed graphs, with several partial successes, see details in Section 2.1.1. However, it is still not known how to solve Max-Flow for multiple source-sink pairs faster than solving it separately for each pair, even in special cases like a single source and all possible sinks. We shall consider the following problems involving

[1]This chapter is based on [KT18b].

Directed	Class	Problem	Running time	Reference
No	General	All-Pairs (G-H Tree)	$(n-1)T(n,m)$	[GH61]
No	Unit-Capacity Networks	All-Pairs (G-H Tree)	$\tilde{O}(mn)$	[KL15], [BHKP07]
No	Genus bounded by g	All-Pairs (G-H Tree)	$2^{O(g^2)}n\log^3 n$	[BENW16]
Yes	Sparse	All-Pairs	$O(n^2+\gamma^4\log\gamma)$	[ACZ98]
Yes	Constant Treewidth	All-Pairs	$O(n^2)$	[ACZ98]
Yes	Unit-Capacity Networks	All-Pairs	$O(m^\omega)$	[CLL13]
Yes	Unit-Capacity DAGs	Single-Source	$O(n^{\omega-1}m)$	[CLL13]
Yes	Planar	Single-Source	$O(n\log^3 n)$	[LNSW12]

Table 2.1: Known algorithms for multiple-pairs Max-Flow. In this table, $T(n,m)$ is the fastest time to compute maximum st-flow in an undirected graph, ω is the matrix multiplication exponent, and $\gamma=\gamma(G)$ is a topological property of the input network that varies between 1 and $\Theta(n)$. In planar graphs, γ is the minimum number of faces required to cover all the nodes (i.e., every node is adjacent to at least one such face) over all possible planar embeddings [Fre95].

multiple source-sink pairs, where the goal is always to report the value of each flow (and not an actual flow attaining it).

Definition 2.1.1. *(All-Pairs Max-Flow) Given a directed edge-capacitated graph $G=(V,E,w)$, output, for every pair of nodes $u,v\in V$, the maximum flow that can be shipped in G from u to v.*

Definition 2.1.2. *(ST-Max-Flow) Given a directed edge-capacitated graph $G=(V,E,w)$ and two subsets of nodes $S,T\subseteq V$, output, for every pair of nodes $s\in S$ and $t\in T$, the maximum flow that can be shipped in G from s to t.*

Definition 2.1.3. *(Global Max-Flow) Given a directed edge capacitated graph $G=(V,E,w)$, output the maximum among all pairs $u,v\in V$, of the maximum flow value that can be shipped in G from u to v.*

Definition 2.1.4. *(Maximum Local Edge Connectivity) Given a directed graph $G=(V,E)$, output the maximum among all pairs $u,v\in V$, of the maximum number of edge-disjoint uv-paths in G.*

Note that in a graph with all edge capacities equal to 1, the problem of finding the maximum local edge connectivity is equivalent to finding the global maximum flow.

2.1.1 Prior Work

We start with undirected graphs, where the All-Pairs Max-Flow values can be represented in a very succint manner, called nowdays *a Gomory-Hu* tree [GH61]. In addition to being very succint, it allows the flow values and the corresponding cuts (vertex partitions) to be quickly retrieved. For a list of previous algorithms for multiple pairs maximum st-flow, see Table 2.1. For directed graphs, no current algorithm computes the maximum flow between any $k=\omega(1)$ given pairs of nodes faster than the time of $O(k)$ separate Max-Flow computations. However, some results are known in special settings. It is possible to compute Max-Flow for $O(n)$ pairs in the time it takes for a single Max-Flow computation [HO94] and this result is used to find a global minimum cut. However, these pairs cannot be specified in the input.

For directed planar graphs, there is an $O(n \log^3 n)$ time algorithm for the Single-Source Max-Flow problem [LNSW12], which immediately yields an $O(n^2 \log^3 n)$ time algorithm for the All-Pairs version, that is much faster than the time of $O(n^2)$ computations of planar Max-Flow, a problem that can be solved in time $O(n \log n)$ [BK09]. Based on these results, it was conjectured in [LNSW12] that also in general graphs, All-Pairs Max-Flow can be solved faster than the time required for computing $O(n^2)$ separate maximum *st*-flows.

Several hardness results are known for multiple-pairs variants of Max-Flow [AWY18]. For ST-Max-Flow in sparse graphs ($m = O(n)$) and $|S| = |T| = O(\sqrt{n})$, there is an $n^{3/2-o(1)}$ lower bound assuming at least one of the Strong Exponential Time Hypothesis (SETH), 3SUM, and All-Pairs Shortest-Paths (APSP) conjectures is correct (for comprehensive surveys on them, see [Vas15, Vas18]). In addition, they show that Single-Source Max-Flow on sparse graphs requires $n^{2-o(1)}$ time, unless MAX-CNF-SAT can be solved in time $2^{(1-\delta)n} \operatorname{poly}(m)$ for some fixed $\delta > 0$, and in particular SETH is false.

We will rely on SETH, a conjecture introduced by [IP01], and on some weaker assumption related to its maximization version, MAX-CNF-SAT. In more detail, SETH states that for every fixed $\varepsilon > 0$ there is an integer $k \geq 3$ such that kSAT on n variables and m clauses cannot be solved in time $2^{(1-\varepsilon)n} \operatorname{poly}(m)$, where $\operatorname{poly}(m)$ refers to $O(m^c)$ for unspecified constant c. By the sparsification lemma [IPZ01], in order to refute SETH it can be assumed that the number of clauses is $O(n)$. The MAX-CNF-SAT problem asks for the maximum number of clauses that can be satisfied in an input CNF formula. Most of our conditional lower bounds are based on the assumption that for every fixed $\delta > 0$, MAX-CNF-SAT cannot be solved in time $2^{(1-\delta)n} \operatorname{poly}(m)$, where currently even $2^n / \operatorname{poly}(n)$ algorithms are not known for this problem [AWY18]. Note that this is a weaker assumption than SETH, since a faster algorithm for MAX-CNF-SAT would imply a faster algorithm for CNF-SAT and refute SETH. Different assumptions regarding the hardness of CNF-SAT have been the basis for many lower bounds, including for the running time of solving NP-hard problems exactly, parametrized complexity, and problems in P. See the Introduction in [ABHS17] and the references therein.

2.1.2 Our Contribution

We present conditional running time lower bounds for both unit and general capacities networks. The proofs appear in sections 2.2 and 2.3, respectively, where the order reflects increasing level of complication. All our lower bounds hold even when the input G is a DAG and has a constant diameter, and in the case of general capacities, they can be easily modified to apply also for graphs with constant maximum degree. In addition, for integer $k \geq 1$ we use $[k]$ to denote the range $\{1, ..., k\}$.

Capacitated Networks Our main result is that for every set sizes $|S|$ and $|T|$, the ST-Max-Flow cannot be solved significantly faster than $O(|S||T|m)$ (i.e., polynomially smaller running time), unless a breakthrough in MAX-CNF-SAT is achieved, and consequently in SETH.

Theorem 2.1.5. *If for some fixed constants $\varepsilon > 0$, $c_1, c_2 \in [0,1]$, ST-Max-Flow on graphs with n nodes, $|S| = \tilde{O}(n^{c_1})$, $|T| = \tilde{O}(n^{c_2})$, $m = O(n)$ edges, and capacities in $[n]$ can be solved* *in time $O((|S||T|m)^{1-\varepsilon})$, then for some $\delta(\varepsilon) > 0$, MAX-CNF-SAT on n' variables and $O(n')$ clauses can be solved in time $2^{(1-\delta)n'} \operatorname{poly}(n')$, and in particular SETH is false.*

This result improves the aforementioned $n^{3/2-o(1)}$ lower bound of [AWY18], as for their setting of $|S| = |T| = O(\sqrt{n})$ our lower bound is $n^{2-o(1)}$, although their lower bound is based on an incomparable (and intuitively weaker) conjecture, that at least one of the SETH, 3SUM, and APSP conjectures is correct. In fact, if there was a reduction from SETH that implied a larger running time lower bound for ST-Max-Flow, then the (single-pair) Max-Flow problem would require a strictly super-linear time under it, but such a reduction is not possible unless the non-deterministic version of SETH (abbreviated NSETH) is false [CGI+16]. And anyway, such a lower bound for Max-Flow would be an amazing breakthrough.

The next theorem is an immediate corollary of Theorem 2.1.5, by assigning $|S|, |T| = \Theta(n)$.

Theorem 2.1.6. *If for some fixed $\varepsilon > 0$, All-Pairs Max-Flow in graphs with n nodes, $m = O(n)$ edges, and capacities in $[n]$ can be solved in time $O((n^2 m)^{1-\varepsilon})$, then for some $\delta(\varepsilon) > 0$, MAX-CNF-SAT on n' variables and $O(n')$ clauses can be solved in time $2^{(1-\delta)n'} \operatorname{poly}(n')$, and in particular SETH is false.*

This conditional lower bound (see Figure 2.1) shows that All-Pairs Max-Flow requires time that is equivalent to $\Omega(n^{3/2})$ computations of Max-Flow, which strongly separates the directed case from the undirected one (where a Gomory-Hu tree can be constructed in the time of $n-1$ computations). If Max-Flow takes $\tilde{O}(m)$ time, which is currently open but plausible, then the running time of $\tilde{\Omega}(n^2)$ computations of Max-Flow is needed. This is in contrast to the aforementioned conjecture of Lacki, Nussbaum, Sankowski, and Wulf-Nilsen [LNSW12] that All-Pairs Max-Flow in general graphs can be solved faster than the time of $O(n^2)$ computations of maximum st-flow.

Unit-Capacity Networks For the case of unit-capacity networks, we show that for every $m = m(n)$, All-Pairs Max-Flow cannot be solved significantly faster than $O(mn)$. Here we introduce a new technique to design reductions from SETH to graphs with varying edge densities, rather than the usual reductions that only deal with sparse graphs.

Theorem 2.1.7. *If for some fixed $\varepsilon > 0$ and $c \in [1, 2]$, All-Pairs Max-Flow in unit-capacity graphs with n nodes and $m = \Theta(n^c)$ edges can be solved in time $O((mn^{1-\varepsilon}))$, then for some $\delta(\varepsilon) > 0$, MAX-CNF-SAT on n' variables and $O(n')$ clauses can be solved in time $2^{(1-\delta)n'} \operatorname{poly}(n')$, and in particular SETH is false.*

Hence, a certain additional improvement to the $O(m^\omega)$ time algorithm of [CLL13] (and similarly to the $O(n^\omega m)$ time for DAGs, where our lower bounds apply too) is not likely. We now present conditional lower bounds for ST-Max-Flow, which are functions of $|S|$ and $|T|$.

Theorem 2.1.8. *If for some fixed constants $\varepsilon > 0$, $c_1, c_2 \in [0, 1]$, ST-Max-Flow on unit-capacity graphs with n nodes, $|S| = \tilde{O}(n^{c_1})$, $|T| = \tilde{O}(n^{c_2})$, and $O((|S| + |T|)n)$ edges can be solved in time $O((|S||T|n)^{1-\varepsilon})$, then for some $\delta(\varepsilon) > 0$, MAX-CNF-SAT on n' variables and $O(n')$ clauses can be solved in time $2^{(1-\delta)n'} \operatorname{poly}(n')$, and in particular SETH is false.*

In addition, we present a conditional lower bound for computing the Maximum Local Edge Connectivity of sparse graphs, which is the same as Global Max-Flow if all the capacities are 1, that is indeed the case in our reduction. The next result, proved in Section 2.5, was obtained together with Bundit Laekhanukit and Rajesh Chitnis, and we thank them for their permission to include it here.

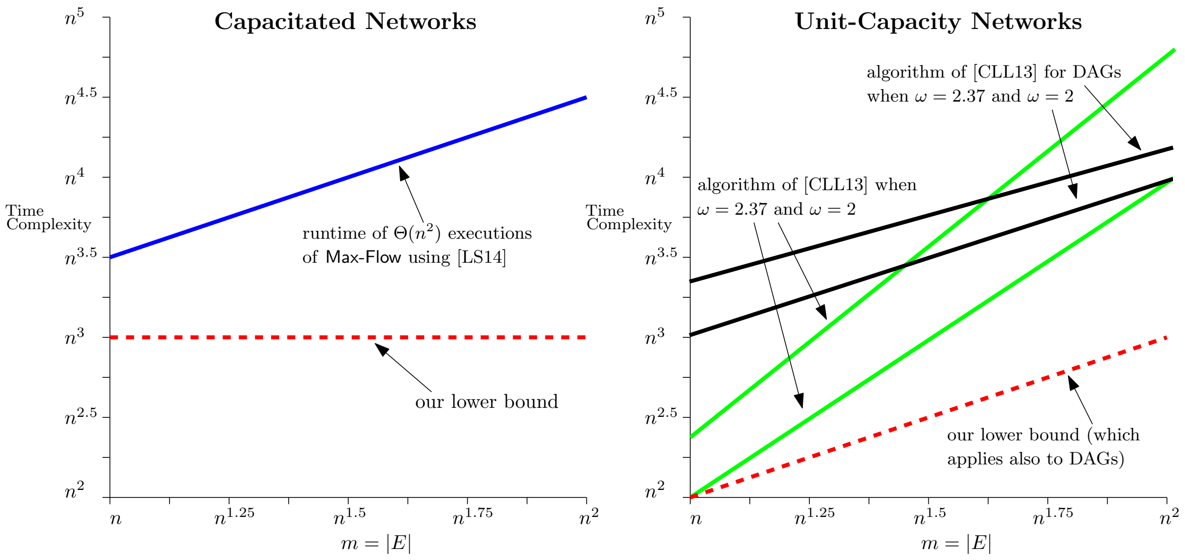

Figure 2.1: State of the art bounds for All-Pairs Max-Flow in directed networks. Conditional lower bounds are depicted in dashed lines, and known algorithms in solid lines.

Theorem 2.1.9. *If for some fixed $\varepsilon > 0$, the Maximum Local Edge Connectivity in graphs with n nodes and $\tilde{O}(n)$ edges can be found in time $O(n^{2-\varepsilon})$, then for some $\delta(\varepsilon) > 0$, MAX-CNF-SAT on n' variables and $O(n')$ clauses can be solved in time $2^{(1-\delta)n'} \operatorname{poly}(n')$, and in particular SETH is false.*

Generalization to Bounded Cuts Finally, we show in Section 2.4 that our lower bounds extend to the version that requires to output the maximum-flow value only for source-sink pairs for which this value is at most some given threshold k.

Connection to the Orthogonal Vectors Problem Our techniques are based on partitioning the variable set of CNF-SAT to sets of different sizes, and constructing graphs with the property that certain pairs of nodes would have smaller maximum flow between them if and only if they correspond to a satisfying assignment. This approach is inspired by results of Williams [Wil05].

We remark that all of our theorems can also be proved assuming that for the appropriate $k \in \{2, 3\}$, the k-Orthogonal Vectors (kOV) problem cannot be solved in time $\tilde{O}(n^{k-\varepsilon})$ for a fixed constant $\varepsilon > 0$, in what is called the kOV Hypothesis (see [Vas15, Vas18]). In the kOV problem the input is k sets $\{U_i\}_{i\in[k]}$, each of n vectors from $\{0,1\}^d$, and the goal is to find k vectors $\{u_i\}_{i\in[k]}$, one from each set, such that $u_1 \cdot \ldots \cdot u_k := \sum_{i=1}^{d} \prod_{j=1}^{k} u_j[i] = 0$ (for $k = 2$ it means that u_1, u_2 are orthogonal). An equivalent version of the problem has $U_1 = \ldots = U_k$. Solving kOV in time $O(n^k d)$ can be done easily by exhaustive search, while the fastest known algorithm for the problem runs in time $n^{k-1/\Theta(\log(d/\log n))}$ [AWY15, CW16]. Williams [Wil05] proved that SETH implies the non-existence of an $\tilde{O}(n^{k-\varepsilon})$-time algorithm.

2.2 Reduction to Multiple-Pairs Max-Flow with Unit Capacity

In this section we prove Theorems 2.1.7 and 2.1.8. We start with a general lemma which is the heart of the proofs.

Lemma 2.2.1. *Let $a \in [0,1]$ and $b \in [0,1-a]$. Then MAX-CNF-SAT on n variables and m clauses $\{C_i\}_{i\in[m]}$ can be reduced to $O(m)$ instances of ST-Max-Flow with $|S| = 2^{an}$ and $|T| = 2^{bn}$ in graphs with $\Theta(2^{an} + 2^{(1-a-b)n}m + 2^{bn})$ nodes, $\Theta((2^{an} + 2^{bn}) \cdot 2^{(1-a-b)n}m)$ edges, and capacities in $\{0,1\}$.*

Proof. Given a CNF-formula F on n variables and m clauses as input for MAX-CNF-SAT, $a \in [0,1]$, and $b \in [0,1-a]$, we split the variables into three sets U_1, U_2, and U_3, where U_1 is of size an, U_2 is of size $(1-a-b)n$, and U_3 is of size bn, and enumerate all their 2^{an}, $2^{(1-a-b)n}$, and 2^{bn} partial assignments (with respect to F), respectively, when the objective is to find a triple (α, β, γ) of assignments to U_1, U_2, and U_3 respectively, that satisfies the maximal number of clauses. We will have an instance G_p of ST-Max-Flow for each value $p \in [m]$, in which by one call to ST-Max-Flow we check if there exists a triple α, β, and γ that satisfies at least p clauses, as follows.

We construct a graph G_p for every $p \in [m]$ on N nodes $V_1 \cup V_2 \cup V_3$, where V_1 contains a node α for every assignment α to U_1, V_2 contains $2m + 1 + (p-1) = 2m + p$ nodes for every assignment β to U_2, that are β_i^l and β_i^r for every $i \in [m]$, β', and the set $\{\beta_i'\}_{i\in[p-1]}$, and V_3 contains a node γ for every assignment γ to U_3. We use the notation α for nodes in V_1 and for assignments to U_1, β for assignments to U_2, and γ for nodes in V_3 and assignments to U_3. However, it will be clear from the context. Now, we have to describe the edges in the network. In order to simplify the reduction, we partition the edges into blue and red colors, as follows.

For every α, β, and $i \in [m]$, we add a blue edge from α to β_i^l if both of α and β do not satisfy the clause C_i (do not set any of the literals to true), and otherwise we add a red edge from α to β_i^r. We further add, for every β, γ, and $i \in [m]$, a blue edge from β_i^l to γ if γ does not satisfy C_i. For every β, γ, and $j \in [p-1]$, we add a red edge from every β_j' to every γ. For every β and $i \in [m]$, we add a red edge from β_i^l to β_i^r and from β_i^r to β', and finally for every β and $j \in [p-1]$, we add a red edge from β' to β_j', where all edges are of capacity 1.

The graph we built has $2^{an} + 2 \cdot 2^{(1-a-b)n}m + 2^{(1-a-b)n} + 2^{(1-a-b)n}(p-1) + 2^{bn} = \Theta(2^{an} + 2^{(1-a-b)n}m + 2^{bn})$ nodes, $2^{an} \cdot 2^{(1-a-b)n}m + 2^{bn} \cdot 2^{(1-a-b)n}m + 2 \cdot 2^{(1-a-b)n}m + (p-1)2^{(1-a-b)n} + 2^{bn} \cdot (p-1)2^{(1-a-b)n} = \Theta((2^{an} + 2^{bn}) \cdot 2^{(1-a-b)n}m)$ edges, with capacities in $\{0,1\}$ (see Figure 2.2), and its construction time is asymptotically the same as the time it takes to output its edge set.

For every α, β, and γ, we denote by $G_p^{\alpha,\beta,\gamma}$ the graph induced from G_p on the nodes

$$(\alpha, \beta', \gamma) \cup \left(\bigcup_{\substack{y\in\{l,r\} \\ i\in[m]}} \beta_i^y \right) \cup \left(\bigcup_{j\in[p-1]} \beta_j' \right)$$

We claim that for every α and γ, the maximum flow from α to γ can be bounded by the sum, over all β, of the maximum flow between them in $G_p^{\alpha,\beta,\gamma}$. This claim follow easily because the intersection $G_p^{\alpha,\beta_1,\gamma} \cap G_p^{\alpha,\beta_2,\gamma}$ for $\beta_1 \neq \beta_2$ is exactly the source and the sink $\{\alpha, \gamma\}$, no edge passes between these two graphs, and $\left(\bigcup_\beta G_i^{\alpha,\beta,\gamma}\right)$ consists of all nodes that are both reachable from α and γ is reachable from them.

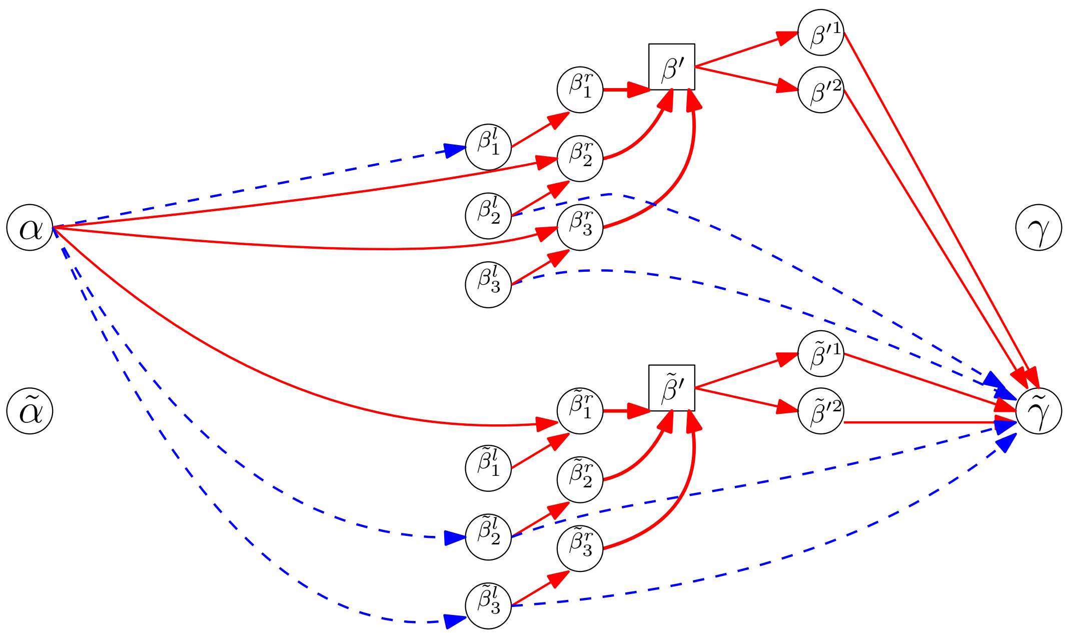

Figure 2.2: An illustration of part of the reduction. Here, U_1, U_2, and U_3 have 2 assignments each, α and $\tilde{\alpha}$ to U_1, β and $\tilde{\beta}$ to U_2, and γ and $\tilde{\gamma}$ to U_3. Blue edges are dashed. For simplicity, only the edges of $G_3^{\alpha,\beta,\tilde{\gamma}} \cup G_3^{\alpha,\tilde{\beta},\tilde{\gamma}}$ are presented. In this illustration, α does not satisfy anything, β satisfies C_2 and C_3, $\tilde{\beta}$ satisfies C_1, and $\tilde{\gamma}$ satisfies C_1. Note that the assignment comprised of α, β, and $\tilde{\gamma}$ satisfies all the clauses, and indeed the maximum flow from α to γ is $2 \cdot 3 - 1 = 5$.

We now prove that if there is an assignment to F that satisfies at least p clauses then the graph G_p we built has a triple α, β, γ with maximum flow from α to γ in $G_p^{\alpha,\beta,\gamma}$ at most $m-1$. Since for every $\tilde{\beta}$, m is the number of outgoing edges from α in $G_p^{\alpha,\tilde{\beta},\gamma}$, m is also an upper bound for the maximum flow from α to γ in it, and hence in G_p it is at most $2^{(1-a-b)n}m - 1$. Otherwise, we will show that every triple α, β, γ has a maximum flow from α to γ in $G_p^{\alpha,\beta,\gamma}$ of size at least m, and so in G_p it is at least $2^{(1-a-b)n}m$. Hence, by simply picking the maximal $j \in [m]$ such that the maximum flow in G_j of some pair α, γ is at most $2^{(1-a-b)n}m - 1$, and then by iterating over all assignments β to U_2 with α and γ fixed as the assignments to U_1 and U_3, we can also find the required triple α, β, γ.

For the first direction, assume that F has an assignment that satisfies at least p clauses, and denote such assignment by Φ. Let α_Φ, β_Φ, and γ_Φ be the assignments to U_1, U_2, and U_3, respectively, that are induced from Φ. Since a blue path from α_Φ through ${\beta_\Phi}^l_i$ for some $i \in [m]$ to γ_Φ corresponds to α_Φ, β_Φ, and γ_Φ all do not satisfy C_i, in $G_p^{\alpha_\Phi,\beta_\Phi,\gamma_\Phi}$ there are at most $m-p$ (internally) disjoint blue paths from α to γ. As the only way to ship flow in $G_p^{\alpha_\Phi,\beta_\Phi,\gamma_\Phi}$ that is not through a blue path is through the node β'_Φ, and the total number of edges going out of this node is $p-1$, we conclude that the total maximum flow in $G_p^{\alpha_\Phi,\beta_\Phi,\gamma_\Phi}$ from α_Φ to β_Φ is bounded by $m-p+(p-1) = m-1$. Since for every β, the maximum amount of flow that can be shipped in $G_p^{\alpha_\Phi,\beta,\gamma_\Phi}$ from α_Φ to γ_Φ is at most m, summing over all β we get that the total flow in G_p from α_Φ to γ_Φ is bounded by $(2^{(1-a-b)n}-1)m+(m-1) \leq 2^{(1-a-b)n}m-1$, as required.

For the second direction, assume that every assignment to F satisfies at most $p-1$ clauses.

In order to show that the maximum flow from every α to every γ is at least $2^{(1-a-b)n}m$, we first fix α, β, and γ. Then, by passing flow in two phases we show that m units of flow can be passed in $G_p^{\alpha,\beta,\gamma}$ from α to γ. As this argument applies for every β, we can add up the respective flows without violating capacities, concluding the proof. By the assumption, there exist $m-(p-1) = m-p+1$ i's, such that α, β, and γ do not satisfy C_i, and we denote a set with this amount of such i's by I_β. Each of these i's induces a blue path $(\alpha \to \beta_i^l \to \gamma)$ from α to γ in $G_p^{\alpha,\beta,\gamma}$, and so we ship a unit of flow through every one of them according to I_β, in what we call the first phase. In the second phase, we ship additional $m-(m-p+1) = p-1$ units in the following way. Let $A_1 := \{i \in [m] \setminus I_\beta : \alpha \nvDash C_i \wedge \beta \nvDash C_i\}$, and $A_2 := ([m] \setminus I_\beta) \setminus A_1 = \{i \in [m] \setminus I_\beta : \alpha \vDash C_i \vee \beta \vDash C_i\}$, where $\alpha \vDash C_i$ denotes that the assignment α satisfies C_i (as defined earlier), and $\alpha \nvDash C_i$ denotes that it does not satisfy C_i. Let $f : A_1 \cup A_2 \to [m-|I_\beta|]$ be a bijective function such that the range of A_1 is $[|A_1|]$ and the range of A_2 is $[m-|I_\beta|] \setminus [|A_1|]$. Clearly, there exists such bijection and it is easy to find one. For every $i \in A_1$ we ship flow through the path $(\alpha \to \beta_i^l \to \beta_i^r \to \beta' \to \beta_j' \to \gamma)$, and for every $i \in A_2$ through the path $(\alpha \to \beta_i^r \to \beta' \to \beta_j' \to \gamma)$, in both cases with $j = f(i)$.

Since we defined the flow in paths, we only need to show that the capacity requirements hold, and we start with blue edges. Indeed, edges of the form (α, β_i^l) are used in the first phase, with flow that is determined uniquely by β and $i \in I_\beta$, and in the second phase uniquely according to β and $i \in [m] \setminus I_\beta$, and so they cannot be used twice. Edges of the form (β_i^l, γ) are only used in the first phase, and their flow is uniquely determined according to β and $i \in I_\beta$, and so are good too. We now proceed to red edges, which were used only in the second phase.

Edges of the forms (α, β_i^r), (β_i^l, β_i^r) and (β_i^r, β') have flow that is uniquely determined by β and $i \in [m] \setminus I_\beta$, and so are not used more than once. Edges of the form (β', β_j') have flow that is uniquely determined by β and $j = f(i) \in [p-1]$, and since f is a bijection, every j has at most one i such that $f(i) = j$, and so these edges are also used at most once. As a byproduct, and since every edge of the form (β_j', γ) has only the edge (β', β_j') as its source for flow, edges of the form (β_j', γ) are also used at most once. Altogether, we have bounded the total flow in all edges that were used in both phases, and so the capacity requirements follow, which completes the proof of the second direction and of Lemma 2.2.1. □

Proof of Theorem 2.1.7. We apply Lemma 2.2.1 in as follows. For every setting of $a = b \in [1/3, 1/2]$ we get graphs $G = (V, E, w)$ with $|V| = \Theta(2^{an})$ ($|V| = \Theta(2^{an})m$ if $a = 1/3$) and $|E| = 2^{(1-a)n}m$. Hence, $|E| = O(|V|^{1/a-1})$ and so in order to get any $c \in [1, 2]$ we can pick $a (= b)$ such that additionally $c = 1/a - 1$, and Theorem 2.1.7 follows. □

Proof of Theorem 2.1.8. Here we apply Lemma 2.2.1 a bit differently. For every setting of $a, b \in [0, 1/2]$ such that $1 - a - b \geq \max(a, b)$ we get graphs $G = (V, E, w)$ with $|V| = \Theta(2^{(1-a-b)n}m)$ and $|E| = \Theta((2^{an} + 2^{bn})2^{(1-a-b)n}m)$. Hence, in order to get any $c_1, c_2 \in [0, 1]$, we can pick a, b such that additionally $c_1 = a/(1-a-b)$ and $c_2 = b/(1-a-b)$, and thus $|S| = (|V|/m)^{c_1}$ and $|T| = (|V|/m)^{c_2}$, and so we get our lower bound for $|E| = O((|S|+|T|)|V|)$ and Theorem 2.1.8 follows.

□

2.3 Reduction to Multiple-Pairs Max-Flow in Capacitated Networks

In this section we prove Theorems 2.1.5 and 2.1.6. We proceed to prove our main technical lemma.

Lemma 2.3.1. *Let $a \in [0,1]$ and $b \in [0, 1-a]$. Then MAX-CNF-SAT on n variables and m clauses $\{C_i\}_{i\in[m]}$ can be reduced to $O(m)$ instances of ST-Max-Flow with $|S| = 2^{an}$ and $|T| = 2^{bn}$ in graphs with $N = \Theta(2^{an} + 2^{(1-a-b)n}m + 2^{bn})$ nodes, $O((2^{an} + 2^{(1-a-b)n} + 2^{bn})m) = O(N)$ edges, and with capacities in $[N]$.*

Proof. Given a CNF-formula F on n variables and m clauses as input for MAX-CNF-SAT, $a \in [0,1]$, and $b \in [0, 1-a]$, we begin similarly to before by splitting the variables into three sets U_1, U_2, and U_3 where U_1 is of size an, U_2 is of size $(1-a-b)n$, and U_3 is of size bn, and enumerate all their 2^{an}, $2^{(1-a-b)n}$, and 2^{bn} partial assignments (with respect to F), respectively, when the objective is to find a triple (α, β, γ) of assignments to U_1, U_2, and U_3, that satisfy the maximal number of clauses. We will have an instance G_p of ST-Max-Flow for each value $p \in [m]$, in which by one call to ST-Max-Flow we check if there exists a triple (α, β, γ) that satisfies at least p clauses, as follows.

We construct the graph G_p on N nodes $V_1 \cup V_2 \cup V_3 \cup A \cup B \cup \{v_B\}$, where V_1 contains a node α for every assignment α to U_1, V_2 contains $3m+1$ nodes for every assignment β to U_2, that are β_i^l, β_i^c, β_i^r, for every $i \in [m]$, and β', V_3 contains a node γ for every assignment γ to U_3, A contains two nodes $C_i^{\models}$ and $C_i^{\not\models}$ for every clause C_i, and B contains a node C_i for every clause C_i. We use the notation α for nodes in V_1 and assignments to U_1, β to assignments to U_2, γ for nodes in V_3 and assignments to U_3, and C_i for nodes in B and clauses. However, it will be clear from the context. Now, we have to describe the edges in the network. In order to simplify the reduction, we partition the edges into red and blue colors, as follows.

For every α and $i \in [m]$ we add a red edge of capacity $2^{(1-a-b)n}$ from α to $C_i^{\models}$ if $\alpha \models C_i$, and a blue edge of the same capacity from α to $C_i^{\not\models}$ otherwise. We further add, for every β, a red edge of capacity 1 from $C_i^{\models}$ to β_i^c, a blue edge of capacity 1 from $C_i^{\not\models}$ to β_i^l, a blue edge of capacity 1 from β_i^l to β_i^r if $\beta \not\models C_i$, a red edge of capacity 1 from β_i^c to β', and a blue edge of capacity 1 from β_i^r to C_i. For every β we add a red edge of capacity $p-1$ from β' to v_B. For every γ we add a red edge of capacity $2^{(1-a-b)n}(p-1)$ from v_B to $\gamma \in V_3$, and finally, for every γ and $i \in [m]$ we add a blue edge of capacity $2^{(1-a-b)n}$ from C_i to γ if $\gamma \not\models C_i$.

The graph we built has $N = 2^{an} + 2m + 2^{(1-a-b)n} \cdot 3m + 2^{(1-a-b)n} + 1 + m + 2^{bn} = \Theta(2^{an} + 2^{(1-a-b)n} \cdot m + 2^{bn})$ nodes, at most $2^{an}m + 2^{(1-a-b)n} \cdot 2m + 2^{(1-a-b)n} \cdot 2m + 2^{(1-a-b)n}m + 2^{(1-a-b)n} + 1 + 2^{(1-a-b)n}m + 2^{bn}m = O((2^{an} + 2^{(1-a-b)n} + 2^{bn})m)$ edges, all of its capacities are in $[N]$, and its construction time is $O(Nm)$ (see Figure 2.3).

We proceed to prove that if there is an assignment to F that satisfies at least p clauses then the graph G_p we built has a pair α, γ with maximum flow from α to γ at most $2^{(1-a-b)n}m - 1$, and otherwise, every α, γ has a maximum flow of size at least $2^{(1-a-b)n}m$. Hence, by simply picking the maximal $j \in [m]$ such that the maximum flow in G_j of some pair α, γ is at most $2^{(1-a-b)n}m - 1$, and then by iterating over all assignments β to U_2 with α and γ fixed as the assignments to U_1 and U_3, we can also find the required triple α, β, γ.

For the first direction, assume that F has an assignment that satisfies at least p clauses, and denote such assignment by Φ. Let α_Φ, β_Φ, and γ_Φ be the assignments to U_1, U_2, and U_3, respectively, that are induced from Φ. We will show that there exists an $(\alpha_\Phi, \gamma_\Phi)$ cut whose

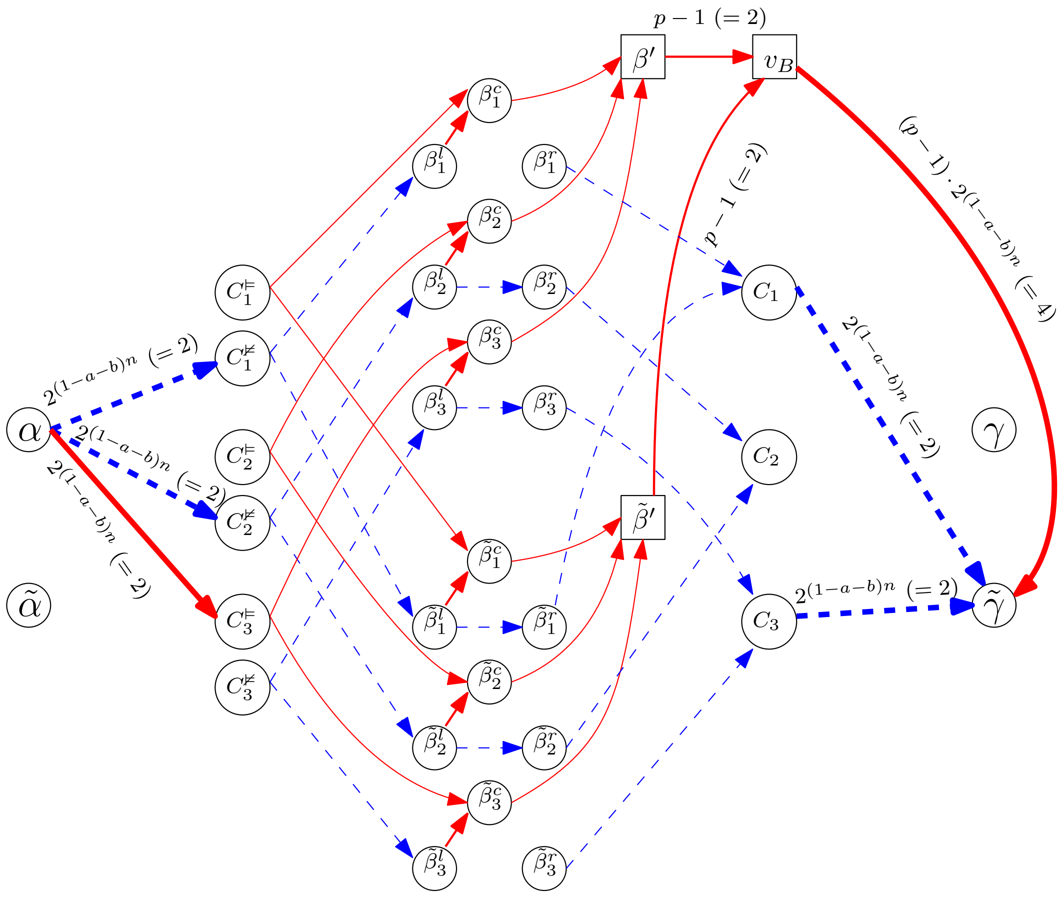

Figure 2.3: An illustration of part of the reduction, with $p = m$. Here, U_1, U_2, and U_3 have 2 assignments each; α and $\tilde{\alpha}$ to U_1, β and $\tilde{\beta}$ to U_2, γ and $\tilde{\gamma}$ to U_3. Bolder edges correspond to edges of higher capacity (specified wherever they are bigger than 1), and blue edges are dashed. For simplicity, only the edges relevant to α and $\tilde{\gamma}$ are presented. In this illustration, α satisfies C_3, β satisfies C_1, $\tilde{\beta}$ satisfies C_3, and $\tilde{\gamma}$ satisfies C_2. Note that the assignment comprised of α, β, and $\tilde{\gamma}$ satisfies all the clauses, and indeed the maximum flow from α to γ is $2 \cdot 3 - 1 = 5$.

---

capacity is at most $2^{(1-a-b)n}m - 1$, hence by the Min-Cut Max-Flow theorem, the maximum flow from α_Φ to γ_Φ is bounded by this number, concluding the proof of the first direction. We define the cut in a way that for every $\beta \neq \beta_\Phi$, the cut will have m cut edges that are contributed from nodes related to β, and nodes related to β_Φ will be carefully added to either side of the cut, so that they will contribute capacity of only $m-1$ to the cut. This is done by exploiting the fact that there are at most $m - p$ blue paths from α_Φ to γ_Φ through nodes associated with β_Φ. To be more precise, we define a suitable cut as follows.

$$S = \{\alpha_\Phi, \beta'_\Phi\} \cup \{C_i^{\models} : \alpha_\Phi \models C_i\} \cup \{C_i^{\not\models} : \alpha_\Phi \not\models C_i\} \cup \{{\beta_\Phi}_i^c : i \in [m]\} \cup \{C_i, {\beta_\Phi}_i^l, {\beta_\Phi}_i^r : \gamma_\Phi \models C_i\} \cup$$
$$\{{\beta_\Phi}_i^l : \gamma_\Phi \not\models C_i \wedge \beta_\Phi \models C_i\}$$

Claim 2.3.2. *The cut $(S, V \setminus S) = (S, T)$ has capacity $2^{(1-a-b)n}m - 1$.*

Proof of Claim. We will go over all the nodes in S, and count the total capacity leaving to nodes in T for each of them. $\alpha_\Phi \in S$ and all nodes $C_i^{\models}$ and $C_i^{\not\models}$ that are adjacent to it are in S too, hence it does not contribute anything. For every $i \in [m]$, we have two cases for nodes in A. If $\alpha_\Phi \models C_i$ then $C_i^{\not\models} \in T$ and hence $C_i^{\not\models}$ does not contribute anything. However, $C_i^{\models}$ has $2^{(1-a-b)n}$ outgoing edges, where all except ${\beta_\Phi}_i^c$ are in T. Hence, it contributes $2^{(1-a-b)n} - 1$ to the cut. Else, if $\alpha_\Phi \not\models C_i$ then $C_i^{\models} \in T$ and hence $C_i^{\models}$ does not contribute anything. But $C_i^{\not\models}$ has $2^{(1-a-b)n}$ outgoing edges, of which $2^{(1-a-b)n} - 1$ are cut edges as their targets are in T, and the one incoming to ${\beta_\Phi}_i^l$ is a cut edge if and only if ${\beta_\Phi}_i \not\models C_i$ and also $\gamma_\Phi \not\models C_i$ (equivalently, ${\beta_\Phi}_i^l \in T$), and in our current case it means that $\Phi \not\models C_i$. Hence, for every $i \in [m]$, the nodes in $\{C_i^{\models}, C_i^{\not\models}\}$ contribute $2^{(1-a-b)n} - 1$ to the cut if $\Phi \models C_i$, and $2^{(1-a-b)n}$ otherwise. Since there are at most $m-p$ clauses that are not satisfied by Φ, summing over all $i \in [m]$ would yield a total of at most $p(2^{(1-a-b)n} - 1) + (m-p)(2^{(1-a-b)n}) = 2^{(1-a-b)n}m - p$ cut edges for vertices with origin in A.

For every $\beta \neq \beta_\Phi$, all nodes in V_2 that are associated with β, v_B, and γ_Φ, are in T and hence will not contribute anything to the cut. However, the node ${\beta_\Phi}'$ is always in S, with v_B its sole target, and hence the edge $({\beta_\Phi}', v_B)$ is in the cut and ${\beta_\Phi}'$ contributes an additional amount of $p-1$, to a current total of at most $2^{(1-a-b)n}m - p + (p-1) = 2^{(1-a-b)n}m - 1$. In addition, ${\beta_\Phi}'$ is the only target of ${\beta_\Phi}_i^c$, and thus ${\beta_\Phi}_i^c$ will not contribute to the cut.

We will show that the rest of the nodes, i.e., nodes in V_2 that are of the forms β_Φ^l and β_Φ^l, and the nodes in B, contribute nothing to the cut. For every $i \in [m]$, ${\beta_\Phi}_i^l \in S$ if and only if either $\beta_\Phi \models C_i$ or $\gamma_\Phi \models C_i$, so we assume that. It always happens that ${\beta_\Phi}_i^c \in S$, and ${\beta_\Phi}_i^r \in T$ if and only if $\gamma_\Phi \not\models C_i$, but in such case, by our assumption it must be that $\beta_\Phi \models C_i$, which implies that the edge $({\beta_\Phi}_i^l, {\beta_\Phi}_i^r)$ is not in the graph, thus the total contribution of ${\beta_\Phi}_i^l$ is zero. Continuing to nodes of the forms ${\beta_\Phi}_i^r$ and C_i, it is easy to verify that the following four statements are either all true or all false: ${\beta_\Phi}_i^r \in S$, $\gamma_\Phi \models C_i$, $C_i \in S$, and the edge (C_i, γ_Φ) is not in the graph. In the case where they all false, in particular C_i and ${\beta_\Phi}_i^r$ are in T and it is clear that they do not contribute anything, so we will focus on the remaining case. Since C_i is in S and is the only target of ${\beta_\Phi}_i^r$, ${\beta_\Phi}_i^r$ will not increase the cut capacity. In addition, since the edge (C_i, γ_Φ) is not in the graph, C_i does not increase the capacity of the cut either. Altogether we have bounded the total capacity of the cut by $2^{(1-a-b)n}m - 1$, finishing the proof of Claim 2.3.2. □

Proceeding with the proof of Lemma 2.3.1, we now focus on the second direction. Assume that every assignment to F satisfies at most $p-1$ clauses. We remind that we need to prove that the maximum flow from every α to every γ is at least $2^{(1-a-b)n}m$, and to do this we first fix α and γ. By the assumption, for every β there exist $m-(p-1) = m-p+1$ i's, such that α, β, and γ do not satisfy C_i, and we denote a set with this amount of such i's by I_β. Each of these i's induces a blue path $(\alpha \to C_i^{\not\models} \to \beta_i^l \to \beta_i^r \to C_i \to \gamma)$ from α to γ, and so we pass a unit of flow through every one of them according to I_β, and for all β, in what we call the first phase. We note that so far, the flow sums up to $2^{(1-a-b)n}(m-p+1)$, and so we carry on with shipping the second phase of flow through paths that are not entirely blue.

We claim that for every β, we can pass an additional amount of $m-(m-p+1) = p-1$ units through β', which would add up to a total flow of $2^{(1-a-b)n}(m-p+1) + 2^{(1-a-b)n}(p-1) = 2^{(1-a-b)n}m$, concluding the proof. Indeed, for every β, we ship flow in the following way. For every $i \in [m] \setminus I_\beta$, if $\alpha \not\models C_i$ then send a unit through $(\alpha \to C_i^{\not\models} \to \beta_i^l \to \beta_i^c \to \beta' \to v_B \to \gamma)$, and otherwise send a unit through $(\alpha \to C_i^{\models} \to \beta_i^c \to \beta' \to v_B \to \gamma)$.

Since we defined the flow in paths, we only need to show that the capacity constraints are satisfied, starting with edges of color blue. Edges of the forms (β_i^l, β_i^r), (β_i^r, C_i), and (C_i, γ) are only used in the first phase, where the flow in the first two is uniquely determined by β and $i \in I_\beta$, and so at most 1 unit of flow is passed through them, and the flow in the latter kind is determined by $i \in I_\beta$, and the same $i \in I_\beta$ can have at most $|\{\beta_i^r\}_\beta| = 2^{(1-a-b)n}$ units of flow passing in (C_i, γ), and so the flow in it is also bounded. The flow in edges of the form $(C_i^{\not\models}, \beta_i^l)$ in the first phase is uniquely determined by β and $i \in I_\beta$, and in the second phase uniquely according to β and $i \in [m] \setminus I_\beta$, and so will not be used twice, and the flow in edges of the form $(\alpha, C_i^{\not\models})$ is determined in the first phase by $i \in I_\beta$ and in the second phase by $i \in [m] \setminus I_\beta$, and so will be used at most $\sum_\beta |I_\beta \cap \{i\}| + \sum_\beta |([m] \setminus I_\beta) \cap \{i\}| \le 2^{(1-a-b)n}$ times.

We now proceed to prove that red edges too do not have more flow than their capacity, and for this we only need to consider the second phase. Edges of the forms $(C_i^{\models}, \beta_i^c)$, (β_i^l, β_i^c), and (β_i^c, β') have flow that is uniquely determined by β and $i \in [m] \setminus I_\beta$ and so are not used more than once, edges of the form (β', v_B) have flow that is determined by β and thus have flow $|\{\beta_i^c\}_{i \in [m] \setminus I_\beta}| = |[m] \setminus I_\beta| = p - 1$, and edges of the form (v_B, γ) have flow of size $(p-1)|\{\beta'\}_\beta| 2^{(1-a-b)n} = (p-1)2^{(1-a-b)n}$, and hence are properly bounded. Finally, edges of the form $(\alpha, C_i^{\models})$ have flow that is determined by $i \in [m] \setminus I_\beta$ and so are used at most $|\{\beta_i^c\}_\beta| = 2^{(1-a-b)n}$ times. Altogether, we have bounded the total flow in all the edges that were used in both phases, and so the capacity requirements follow, which completes the proof of the second direction and of Lemma 2.3.1. □

Proof of Theorem 2.1.5. We apply Lemma 2.3.1 in the following way. For every setting of $a, b \in [0, 1/2]$ such that $1 - a - b \ge \max(a, b)$ we get graphs $G = (V, E, w)$ with $|V| = \Theta(2^{(1-a-b)n} m)$ and $|E| = \Theta(2^{(1-a-b)n} m) = O(|V|)$. Hence, in order to get any $c_1, c_2 \in [0, 1]$, we can pick a, b such that additionally $c_1 = a/(1-a-b)$ and $c_2 = b/(1-a-b)$, and thus $|S| = (|V|/m)^{c_1}$ and $|T| = (|V|/m)^{c_2}$, and so our claimed lower bound and Theorem 2.1.5 follow.

□

2.4 Generalization to Bounded Cuts

Our lower bounds extend to the version where we only care about vertex-pairs with maximum flow bounded by a given k, which we refer to as kPMF.

Definition 2.4.1. *(kPMF) Given a directed edge-capacitated graph $G = (V, E, w)$ and an integer k, for every pair of nodes $u, v \in V$ where the maximum flow that can be shipped in G from u to v is of size at most k, output this pair and its maximum flow value.*

Theorem 2.4.2 (Generalization of Theorem 2.1.7)**.** *If for some fixed constants $\varepsilon > 0$ and $c \in [0, 1]$, kPMF in unit-capacity graphs with n nodes, $k = \tilde{O}(n^c)$, and $m = O(kn)$ edges can be solved in time $O((n^2 k)^{1-\varepsilon})$, then for some $\delta(\varepsilon) > 0$, MAX-CNF-SAT on n' variables and $O(n')$ clauses can be solved in time $2^{(1-\delta)n'} \operatorname{poly}(n')$, and in particular SETH is false.*

Proof. We apply Lemma 2.2.1 as follows. For every setting of $a = b \in [1/3, 1/2]$ we get graphs $G = (V, E, w)$ with $|V| = \Theta(2^{an})$ ($|V| = \Theta(2^{an} m)$ if $a = 1/3$), and $|E| = 2^{an} \cdot 2^{(1-2a)n} m = \Theta(2^{(1-a)n} m)$. The main idea is that the middle layer bound the flow from every α to every γ, which are the only pairs that we need to find the maximum flow for. To be more precise, for every α' and γ' we show a cut of capacity $k = O(2^{(1-2a)n} m)$ separating them, by considering

$$S = \{\alpha'\} \cup \{\beta_i^l : i \in [m], \forall \beta\}.$$

Clearly, the only outgoing edges from S are from α' and from vertices of the form β_i^l. α' has an outgoing degree at most $O(2^{(1-2a)n}m)$, and for each β and $i \in [m]$, vertices of the form β_i^l have a total outgoing degree at most 2. Hence, the total capacity of the cut is bounded by $k = O(2^{(1-2a)n}m)$. The claimed range of k is attained because setting $a = 1/2$ yields $k = O(m) = O(\log|V|) \leq O(n^c)$, and letting a approach 1/3 yields k tending to $O(2^{n/3}m) = O(|V|)$. Note that $|E| = O(|V|k)$, and $|V|^2k = O((2^{an})^2 \cdot 2^{(1-2a)n}m) = O(2^n m)$, and finally in order to get any $c \in [0,1]$ we can pick $a(= b)$ such that additionally $c = 1/a - 2$, and Theorem 2.4.2 holds. □

Theorem 2.4.3 (Generalization of Theorem 2.1.6)**.** *If for some fixed constants $\varepsilon > 0$ and $c \in [0,1]$, kPMF in graphs with n nodes, $k = \tilde{O}(n^c)$, $m = O(n)$ edges, and capacities in $[n]$ can be solved in time $O((n^2k)^{1-\varepsilon})$, then for some $\delta(\varepsilon) > 0$, MAX-CNF-SAT on n' variables and $O(n')$ clauses can be solved in time $2^{(1-\delta)n'}\operatorname{poly}(n')$, and in particular SETH is false.*

Proof. We apply Lemma 2.3.1 in a similar fashion to the application of Lemma 2.2.1 in the proof of Theorem 2.4.2, where the choices of a and b are done in exactly the same way as before, also allowing again a free choice of $c \in [0,1]$. However, now $|E| = O(|V|)$, and we choose the cut as follows. For every α' and γ' we show a cut of capacity $k = O(2^{(1-2a)n}m)$ separating them, by considering

$$S = \{\alpha'\} \cup \{C_i^{\models}, C_i^{\not\models} : i \in [m]\} \cup \{\beta_i^l, \beta_i^c : i \in [m], \forall \beta\}.$$

Clearly, the only outgoing edges from S are of capacity 1, from α' and from vertices of the forms β_i^l and β_i^c. For each β and $i \in [m]$, these vertices have a total of at most 2 edges going out to the rest of the graph. Hence, the size of the cut is bounded by $k = O(2^{(1-2a)n}m)$, and the range of k is similar to the proof of Theorem 2.4.2, and so Theorem 2.4.3 holds.

□

Known algorithms solve kPMF in *directed* graphs in time $\tilde{O}(n^2m \cdot \min(k, \sqrt{n}))$, which is bigger than the lower bound in Theorem 2.4.3 by a factor that is roughly between $\sqrt{n}$ and n for sparse graphs, leaving a gap that is not too big even for relatively small values of k. This running time can be achieved by $O(n^2)$ computations of either the aforementioned $O(mk)$ time algorithm of [FF56] (actually, a slightly modified version that halts when the total flow exceeds k), or the $\tilde{O}(m\sqrt{n})$ time algorithm of [LS14].

It is interesting to note that in graphs that are *undirected* and with unit capacities, an algorithm for kPMF with running time $O(mk + n^2)$ was shown in [BHKP07]. This shows a separation between the directed and the undirected cases also for unit-capacity graphs, roughly by a factor $\Omega(n^2k/(mk+n^2)) = \Omega(\min(k, n/k))$, since our relevant conditional lower bound is proved for $m = O(kn)$. Their algorithm actually builds in time $O(mk)$ a partial Gomory-Hu tree that succinctly represents the values required by kPMF, and then it is easy to extract all the relevant values in time $O(n^2)$, as required by our definition of kPMF. For instance, when $k = O(\sqrt{n})$ and $m = O(n^{3/2})$ their upper bound for the undirected and unit-capacity case is $O(n^2)$, while our lower bound for the directed case is $n^{2.5-o(1)}$.

2.5 Global Max-Flow

Proof of Theorem 2.1.9. Given a CNF-formula F on n variables and m clauses $\{C_i\}_{i\in[m]}$ as input for MAX-CNF-SAT, we split the variables into two sets U_1 and U_2 of size $n/2$ each and enumerate all $2^{n/2}$ partial assignments (with respect to F) to each of them, when the objective is to find a pair (α, β) of assignments to U_1 and U_2 that satisfy the maximal number of clauses. We construct a graph $G = (V, E)$ such that $V = L \cup R \cup C$ as follows. L contains a node α for every assignment α to U_1, R contains a node β for every assignment β to U_2, and C contains three nodes $c_{0,0}$, $c_{0,1}$, and $c_{1,0}$ for every clause C_i. We use the notation α for nodes in L and assignments to U_1, β for nodes in R and assignments to U_2. However, it will be clear from the context. For every assignment α to U_1 and clause C_i, we add an edge from α to $c_{\models,\models}$ and $c_{\models,\not\models}$ if $\alpha \models C_i$, and an edge from α to $c_{\not\models,\models}$ otherwise. Similarly, for every assignment β to U_2 and clause C_i, we add an edge from β to $c_{\models,\models}$ and $c_{\not\models,\models}$ if $\beta \models C_i$, and an edge from β to $c_{\models,\not\models}$ otherwise. This graph has $N = n + n + 3m = O(n)$ nodes and at most $n \cdot 2m + n \cdot 2m = \tilde{O}(n)$ edges. For every pair of assignments α and β and clause C_i there is exactly one path (of length 2) from α to β through nodes associated with C_i if and only if both $\alpha \models C_i$ and $\beta \models C_i$, and no paths through them otherwise. Hence, the number of edge disjoint paths from α to β is exactly the number of clauses that are satisfied by both of the assignments α and β, and so an algorithm for Maximum Local Edge Connectivity with running time $\tilde{O}(n^{2-\varepsilon})$ implies an algorithm for MAX-CNF-SAT with running time $\tilde{O}((2^{n/2})^{2-\varepsilon}) = \tilde{O}(2^{(1-\varepsilon/2)n})$, completing the proof for $\delta(\varepsilon) = \varepsilon/2$. $\square$

Acknowledgements We thank Rajesh Chitnis and Bundit Laekhanukit for some useful conversations, and for their part in achieving the result on Global Max-Flow.

Chapter 3

Faster Algorithms for All-Pairs Bounded Min-Cuts in Unit-Capacity Directed Graphs[1]

3.1 Introduction

Connectivity-related problems are some of the most well-studied problems in graph theory and algorithms, and have been thoroughly investigated in the literature. Given a directed graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges, perhaps the most fundamental such problem is to compute a *minimum st-cut*, i.e., a set of edges E' of minimum-cardinality such that t is not reachable from s in $G \setminus E'$. This minimum st-cut problem is well-known to be equivalent to maximum st-flow, as they have the exact same value [FF62]. Currently, the fastest algorithms for this problem run in time $\tilde{O}(m\sqrt{n}\log^{O(1)} U)$ [LS14] and $\tilde{O}(\min\{m^{10/7}U^{1/7}, m^{11/8}U^{1/4}, m^{4/3}U^{1/3}\})$ (faster for sparse graphs) [Mad16, LS20a, LS20b], where U is the maximum edge capacity (aka weight).[2]

The central problem of study in this chapter is All-Pairs Min-Cut (also known as All-Pairs Max-Flow), where the input is a digraph $G = (V, E)$ and the goal is to compute the minimum st-cut value for all $s, t \in V$. All our graphs will have *unit* edge/vertex-capacities (aka uncapacitated), in which case the value of the minimum st-cut is just the maximum number of disjoint paths from s to t (aka edge/vertex connectivity), by [Men27]. We will consider a few variants: vertex capacities vs. edge capacities,[3] or a general digraph vs. a directed acyclic graph (DAG). For all these variants, we will be interested in the *k-bounded* version (aka *bounded min-cuts*, hence the title of the chapter) where the algorithm needs to find which minimum st-cuts have value less than a given parameter $k < n$, and report only those. Put differently, the goal is to compute, for every $s, t \in V$, the minimum between k and the actual minimum st-cut value. Nonetheless, some of our results (the lower bounds) are of interest even without this restriction.

The time complexity of these problems should be compared against the fundamental

[1]This chapter is based on [AGI+19].

[2]The notation $\tilde{O}(\cdot)$ hides polylogarithmic factors.

[3]The folklore reduction where each vertex v is replaced by two vertices connected by an edge $v_{in} \to v_{out}$ shows that in all our problems, vertex capacities are no harder (and perhaps easier) than edge capacities. Notice that this is only true for directed graphs.

special case that lies at their core — the Transitive Closure problem (aka All-Pairs Reachability), which is known to be time-equivalent to Boolean Matrix Multiplication, and in some sense, to Triangle Detection [WW18]. This is the case $k = 1$, and it can be solved in time $O(\min\{mn, n^\omega\})$, where $\omega < 2.38$ is the matrix-multiplication exponent [CW90, LG14, Vas12]; the latter term is asymptotically better for dense graphs, but it is not *combinatorial*.[4] This time bound is conjectured to be optimal for Transitive Closure, which can be viewed as a conditional lower bound for All-Pairs Min-Cut; but can we achieve this time bound algorithmically, or is All-Pairs Min-Cut a harder problem?

The naive strategy for solving All-Pairs Min-Cut is to execute a minimum *st*-cut algorithm $O(n^2)$ times, with total running time $\tilde{O}(n^2 m^{10/7})$ [Mad16] or $\tilde{O}(n^{2.5} m)$ [LS14]. For not-too-dense graphs, there is a faster randomized algorithm of Cheung, Lau, and Leung [CLL13] that runs in time $O(m^\omega)$. For smaller k, some better bounds are known. First, observe that a minimum *st*-cut can be found via k iterations of the Ford-Fulkerson algorithm [FF62] in time $O(km)$, which gives a total bound of $O(n^2 mk)$. Another randomized algorithm of [CLL13] runs in better time $O(mnk^{\omega-1})$ but it works only in DAGs. Notice that the latter bound matches the running time of Transitive Closure if the graphs are sparse enough. For the case $k = 2$, Georgiadis et al. [GGI$^+$17] achieved the same running time as Transitive Closure up to sub-polynomial factor $n^{o(1)}$ in all settings, by devising two deterministic algorithms, whose running times are $\tilde{O}(mn)$ and $\tilde{O}(n^\omega)$.

Other than the lower bound from Transitive Closure, the main previously known result is from [KT18b], which showed that under the Strong Exponential Time Hypothesis (SETH),[5] All-Pairs Min-Cut requires, up to sub-polynomial factors, time $\Omega(mn)$ in unit-capacity digraphs of any edge density, and even in the simpler case of (unit) vertex-capacities and of DAGs. As a function of k their lower bound becomes $\Omega(n^{2-o(1)}k)$ [KT18b]. Combining the two, we have a conditional lower bound of $(n^2 k + n^\omega)^{1-o(1)}$.

Related Work. There are many other results related to our problem, let us mention a few. Other than DAGs, the problem has also been considered in the special cases of planar digraphs [ACZ98, LNSW12], sparse digraphs and digraphs with bounded treewidth [ACZ98].

In *undirected* graphs, the problem was studied extensively following the seminal work of Gomory and Hu [GH61] in 1961, which introduced a representation of All-Pairs Min-Cuts via a weighted tree, commonly called a Gomory-Hu tree, and further showed how to compute it using $n - 1$ executions of maximum *st*-flow. Bhalgat et al. [BHKP07] designed an algorithm that computes a Gomory-Hu tree in unit-capacity undirected graphs in $\tilde{O}(mn)$ time, and this upper bound was recently improved [AKT20b]. The case of bounded min-cuts (small k) in undirected graphs was studied by Hariharan et al. [HKP07], motivated in part by applications in practical scenarios. The fastest running time for this problem is $\tilde{O}(mk)$ [Pan16], achieved by combining results from [HKP07] and [BHKP07]. On the negative side, there is an $n^{3-o(1)}$ lower bound for All-Pairs Min-Cut in sparse *capacitated* digraphs [KT18b], and very recently, a similar lower bound was shown for *undirected* graphs with vertex capacities [AKT20b].

[4]Combinatorial is an informal term to describe algorithms that do not rely on fast matrix-multiplication algorithms, which are infamous for being impractical. See [AW14, ABW18] for further discussions.

[5]These lower bounds hold even under the weaker assumption that the 3-Orthogonal Vectors problem requires $n^{3-o(1)}$ time.

3.1.1 Our Contribution

The goal of this work is to reduce the gaps in our understanding of the All-Pairs Min-Cut problem (see Table 3.1 for a list of known and new results). In particular, we are motivated by three high-level questions. First, how large can k be while keeping the time complexity the same as Transitive Closure? Second, could the problem be solved in cubic time (or faster) in all settings? Currently no $\Omega(n^{3+\varepsilon})$ lower bound is known even in the hardest settings of the problem (capacitated, dense, general graphs). And third, can the actual cuts (witnesses) be reported in the same amount of time it takes to only report their values? Some of the previous techniques, such as those of [CLL13], cannot do that.

A New Algorithm. Our first result is a randomized algorithm that solves the k-bounded version of All-Pairs Min-Cut in a digraph with *unit vertex-capacities* in time $O((nk)^{\omega})$. This upper bound is only a factor k^{ω} away from that of Transitive Closure, and thus matches it up to polynomial factors for any $k = n^{o(1)}$. Moreover, any poly(n)-factor improvement over our upper bound would imply a breakthrough for Transitive Closure (and many other problems). Our algorithm builds on the network-coding method of [CLL13], and in effect adapts this method to the easier setting of vertex capacities, to achieve a better running time than what is known for unit edge-capacities. This algorithm is actually more general: Given a digraph $G = (V, E)$ with unit vertex-capacities, two subsets $S, T \subseteq V$ and $k > 0$, it computes for all $s \in S, t \in T$ the minimum st-cut value if this value is less than k, all in time $O((n + (|S| + |T|)k)^{\omega} + |S||T|k^{\omega})$. We overview these results in Section 3.3, with full details in Section 3.5.

New Lower Bounds. Finally, we present conditional lower bounds for our problem, the k-bounded version of All-Pairs Min-Cut. As a result, we identify new settings where the problem is harder than Transitive Closure, and provide the first evidence that the problem cannot be solved in cubic time. Technically, the main novelty here is a reduction from the 4-Clique problem. It implies lower bounds that apply to the basic setting of DAGs with unit vertex-capacities, and therefore immediately apply also to more general settings, such as edge capacities, capacitated inputs, and general digraphs, and they in fact improve over previous lower bounds [AWY18, KT18b] in all these settings.[6] We prove the following theorem in Section 3.4.

Theorem 3.1.1. *If for some fixed $\varepsilon > 0$ and any $k \in [n^{1/2}, n]$, the k-bounded version of All-Pairs Min-Cut can be solved on DAGs with unit vertex-capacities in time $O((n^{\omega-1}k^2)^{1-\varepsilon})$, then 4-Clique can be solved in time $O(n^{\omega+1-\delta})$ for some $\delta = \delta(\varepsilon) > 0$.*

Moreover, if for some fixed $\varepsilon > 0$ and any $k \in [n^{1/2}, n]$ that version of All-Pairs Min-Cut can be solved combinatorially in time $O((n^2k^2)^{1-\varepsilon})$, then 4-Clique can be solved combinatorially in time $O(n^{4-\delta})$ for some $\delta = \delta(\varepsilon) > 0$.

To appreciate the new bounds, consider first the case $k = n$, which is equivalent to not restricting k. The previous lower bound, under SETH, is $n^{3-o(1)}$ and ours is larger by a factor of $n^{\omega-2}$. For combinatorial algorithms, our lower bound is $n^{4-o(1)}$, which is essentially the largest possible lower bound one can prove without a major breakthrough

[6]It is unclear if our new reduction can be combined with the ideas in [AKT20b] to improve the lower bounds in the seemingly easier case of undirected graphs with vertex capacities.

in fine-grained complexity. This is because the naive algorithm for All-Pairs Min-Cuts is to invoke an algorithm for Max-Flow $O(n^2)$ times, hence a lower bound larger than $\Omega(n^4)$ for our problem would imply the first non-trivial lower bound for minimum *st*-cut. The latter is perhaps the biggest open question in fine-grained complexity, and in fact many experts believe that near-linear time algorithms for minimum *st*-cut do exist, and can even be considered "combinatorial" in the sense that they do not involve the infamous inefficiencies of fast matrix multiplication. If such algorithms for minimum *st*-cut do exist, then our lower bound is tight.

Our lower bound shows that as k exceeds $n^{1/2-o(1)}$, the time complexity of k-bounded of All-Pairs Min-Cut exceeds that of Transitive Closure by polynomial factors. The lower bound is super-cubic whenever $k \geq n^{2-\omega/2+\varepsilon}$.

Time		Input	Output	Reference
$O(mn), \tilde{O}(n^\omega)$	deterministic	digraphs	cuts, only $k = 2$	[GGI+17]
$O(n^2mk)$	deterministic	digraphs	cuts	[FF62]
$O(m^\omega)$	randomized	digraphs	cut values	[CLL13]
$O(mnk^{\omega-1})$	randomized	digraphs	cut values	[CLL13]
$O((nk)^\omega)$	randomized, vertex cap.	digraphs	cut values	Theorem 3.5.2
$2^{O(k^2)}mn$	deterministic	DAGs	cuts	[AGI+19]
$(k \log n)^{4^k+o(k)} \cdot n^\omega$	deterministic	DAGs	cuts	[AGI+19]
$(mn + n^\omega)^{1-o(1)}$	based on Trans. Closure	DAGs	cut values	
$n^{2-o(1)}k$	based on SETH	DAGs	cut values	[KT18b]
$n^{\omega-1-o(1))}k^2$	based on 4-Clique	DAGs	cut values	Theorem 3.1.1

Table 3.1: Summary of new and known results. Unless mentioned otherwise, all upper and lower bounds hold both for unit edge-capacities and for unit vertex capacitities.

3.2 Preliminaries

We start with some terminology and well-known results on graphs and cuts. Next we will briefly introduce the main algebraic tools that will be used throughout the chapter. We note that although we are interested in solving the k-bounded All-Pairs Min-Cut problem, where we wish to find the all-pairs min-cuts of size at most $k - 1$, for the sake of using simpler notation we compute the min-cuts of size at most k (instead of less than k) solving this way the $(k + 1)$-bounded All-Pairs Min-Cut problem.

Directed graphs. The input of our problem consists of an integer $k \geq 1$ and a *directed graph*, digraph for short, $G = (V, E)$ with $n := |V|$ *vertices* and $m := |E|$ *edges*. All our results extend to multi-digraphs, where each pair of vertices can be connected with multiple (*parallel*) edges. For parallel edges, we always refer to each edge individually, as if each edge had a unique identifier. So whenever we refer to a set of edges, we refer to the set of their unique identifiers, i.e., without collapsing parallel edges, like in a multi-set.

Flows and cuts. We follow the notation used by Ford and Fulkerson [FF62]. Let $G = (V, E)$ be a digraph, where each edge e has a nonnegative capacity $c(e)$. For a pair of vertices s and t, an st-flow of G is a function f on E such that $0 \le f(e) \le c(e)$, and for every vertex $v \ne s, t$ the incoming flow is equal to outgoing flow, i.e., $\sum_{(u,v)\in E} f(u, v) = \sum_{(v,u)\in E} f(v, u)$. If G has vertex capacities as well, then f must also satisfy $\sum_{(u,v)\in E} f(u, v) \le c(v)$ for every $v \ne s, t$, where $c(v)$ is the capacity of v. The value of the flow is defined as $|f| = \sum_{(s,v)\in E} f(s, v)$.

3.3 Overview of Our Algorithmic Approach

In the framework of [CLL13] edges are encoded as vectors, so that the vector of each edge $e = (u, v)$ is a randomized linear combination of the vectors correspond to edges incoming to u, the source of e. One can compute all these vectors for the whole graph, simultaneously, using some matrix manipulations. The bottleneck is that one has to invert a certain $m \times m$ matrix with an entry for each pair of edges. Just reading the matrix that is output by the inversion requires $\Omega(m^2)$ time, since most entries in the inverted matrix are expected to be nonzero even if the graph is sparse.

To overcome this barrier, while using the same framework, we define the encoding vectors on the nodes rather than the edges. We show that this is sufficient for the vertex-capacitated setting. Then, instead of inverting a large matrix, we need to compute the rank of certain submatrices which becomes the new bottleneck. When k is small enough, this turns out to lead to a significant speed up compared to the running time in [CLL13].

3.4 Reducing 4-Clique to All-Pairs Min-Cut

In this section we prove Theorem 3.1.1 by showing new reductions from the 4-Clique problem to k-bounded All-Pairs Min-Cut with unit vertex-capacities. These reductions yield conditional lower bounds that are much higher than previous ones, which are based on SETH, in addition to always producing DAGs. Throughout this section, we will often use the term nodes for vertices.

Definition 3.4.1 (The 4-Clique Problem)**.** *Given a 4-partite graph G, where $V(G) = A \cup B \cup C \cup D$ with $|A| = |B| = |C| = |D| = n$, decide whether there are four nodes $a \in A$, $b \in B$, $c \in C$, $d \in D$ that form a clique.*

This problem is equivalent to the standard formulation of 4-Clique (without the restriction to 4-partite graphs). The currently known running times are $O(n^{\omega+1})$ using matrix multiplication [EG04], and $O(n^4/\operatorname{polylog} n)$ combinatorially [Yu18]. The k-Clique Conjecture [ABW18] hypothesizes that current clique algorithms are optimal. Usually when the k-Clique Conjecture is used, it is enough to assume that the current algorithms are optimal for every k that is a multiple of 3, where the known running times are $O(n^{\omega k/3})$ [NP85] and $O(n^k/\operatorname{polylog} n)$ combinatorially [Vas09], see e.g. [ABBK17, ABW18, BW17, Cha15, LWW18]. However, we will need the stronger assumption that one cannot improve the current algorithms for $k = 4$ by any polynomial factor. This stronger form was previously used by Bringmann, Grønlund, and Larsen [BGL17].

3.4.1 Reduction to the Unbounded Case

We start with a reduction to the unbounded case (equivalent to $k = n$), that is, we reduce to All-Pairs Min-Cut with unit node-capacities (abbreviated APMVC, for All-Pairs Minimum Vertex-Cut). Later (in Section 3.4.1) we will enhance the construction in order to bound k.

Lemma 3.4.2. *Suppose APMVC on n-node DAGs with unit node-capacities can be solved in time $T(n)$. Then 4-Clique on n-node graphs can be solved in time $O(T(n) + MM(n, n))$, where $MM(n, n)$ is the time to multiply two matrices from $\{0, 1\}^{n \times n}$.*

To illustrate the usage of this lemma, observe that an $O(n^{3.99})$-time combinatorial algorithm for APMVC would imply a combinatorial algorithm with similar running time for 4-Clique.

Proof. Given a 4-partite graph G as input for the 4-Clique problem, the graph H is constructed as follows. The node set of H is the same as G, and we abuse notation and refer also to $V(H)$ as if it is partitioned into A,B,C, and D. Thinking of A as the set of sources and D as the set of sinks, the proof will focus on the number of node-disjoint paths from nodes $a \in A$ to nodes $d \in D$. The edges of H are defined in a more special way, see also Figure 3.1 for illustration.

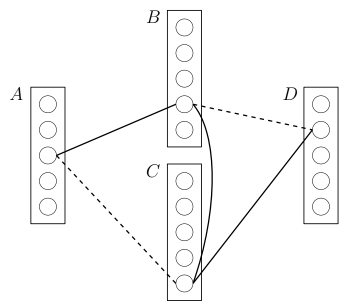

Figure 3.1: An illustration of H in the reduction. Solid lines between nodes represent the existence of an edge in the input graph G, and dashed lines represent the lack thereof.

- (A to B) For every $a \in A, b \in B$ such that $\{a, b\} \in E(G)$, add to $E(H)$ a directed edge (a, b).
- (B to C) For every $b \in B, c \in C$ such that $\{b, c\} \in E(G)$, add to $E(H)$ a directed edge (b, c).
- (C to D) For every $c \in C, d \in D$ such that $\{c, d\} \in E(G)$, add to $E(H)$ a directed edge (c, d).

The definition of the edges of H will continue shortly. So far, edges in H correspond to edges in G, and there is a (directed) path $a \to b \to c \to d$ if and only if the three (undirected) edges $\{a, b\}, \{b, c\}, \{c, d\}$ exist in G. In the rest of the construction, our goal is to make this 3-hop path contribute to the final $a \to d$ flow *if and only if* (a, b, c, d) is a 4-clique in G (i.e., all six edges exist, not only those three). Towards this end, additional edges are introduced, that make this 3-hop path useless in case $\{a, c\}$ or $\{b, d\}$ are not also edges in G. This allows "checking" for five of the six edges in the clique, rather than just three. The sixth edge is easy to "check".

- (A to C) For every $a \in A, c \in C$ such that $\{a, c\} \notin E(G)$, add to $E(H)$ a directed edge (a, c).
- (B to D) For every $b \in B, d \in D$ such that $\{b, d\} \notin E(G)$ in G, add to $E(H)$ a directed edge (b, d).

This completes the construction of H. Note that these additional edges imply that there is a path $a \to b \to d$ in H iff $\{a, b\} \in E(G)$ and $\{b, d\} \notin E(G)$, and similarly, there is a path $a \to c \to d$ in H iff $\{a, c\} \notin E(G)$ and $\{c, d\} \in E(G)$. Let us introduce notations to capture these paths. For nodes $a \in A, d \in D$ denote:

$$B'_{a,d} = \{b \in B \mid \{a, b\} \in E(G) \text{ and } \{b, d\} \notin E(G)\},$$
$$C'_{a,d} = \{c \in C \mid \{a, c\} \notin E(G) \text{ and } \{c, d\} \in E(G)\}.$$

We now argue that if an APMVC algorithm is run on H, enough information is received to be able to solve 4-Clique on G by spending only an additional post-processing stage of $O(n^3)$ time.

Claim 3.4.3. *Let $a \in A, d \in D$ be nodes with $\{a, d\} \in E(G)$. If the edge $\{a, d\}$ does not participate in a 4-clique in G, then the node connectivity from a to d in H, denoted $NC(a, d)$, is exactly*

$$NC(a, d) = |B'_{a,d}| + |C'_{a,d}|,$$

and otherwise $NC(a, d)$ is strictly larger.

Proof of Claim 3.4.3. We start by observing that all paths from a to d in H have either two or three hops.

Assume now that there is a 4-clique (a, b^*, c^*, d) in G, and let us exhibit a set P of node-disjoint paths from a to d of size $|B'_{a,d}| + |C'_{a,d}| + 1$. For all nodes $b \in B'_{a,d}$, add to P the 2-hop path $a \to b \to d$. For all nodes $c \in C'_{a,d}$, add to P the 2-hop path $a \to c \to d$. So far, all these paths are clearly node-disjoint. Then, add the 3-hop path $a \to b^* \to c^* \to d$ to P. This path is node-disjoint from the rest because $b^* \notin B'_{a,d}$ (because $\{b^*, d\} \in E(G)$) and $c^* \notin C'_{a,d}$ (because $\{a, c^*\} \in E(G)$).

Next, assume that no nodes $b \in B, c \in C$ complete a 4-clique with a, d. Then for every set P of node-disjoint paths from a to d, there is a set P' of 2-hop node-disjoint paths from a to d that has the same size. To see this, let $a \to b \to c \to d$ be some 3-hop path in P. Since (a, b, c, d) is not a 4-clique in G and $\{a, d\}, \{a, b\}, \{b, c\}, \{c, d\}$ are edges in G, we conclude that either $\{a, c\} \notin E(G)$ or $\{b, d\} \notin E(G)$. If $\{a, c\} \notin E(G)$ then $a \to c$ is an edge in H and the 3-hop path can be replaced with the 2-hop path $a \to c \to d$ (by skipping b) and one is remained with a set of node-disjoint paths of the same size. Similarly, if $\{b, d\} \notin E(G)$ then $b \to d$ is an edge in H and the 3-hop path can be replaced with the 2-hop path $a \to b \to d$. This can be done for all 3-hop paths and result in P'. Finally, note that the number of 2-hop paths from a to d is exactly $|B'_{a,d}| + |C'_{a,d}|$, and this completes the proof of Claim 3.4.3. □

Computing the estimates. To complete the reduction, observe that the values $|B'_{a,d}| + |C'_{a,d}|$ can be computed for all pairs $a \in A, d \in D$ using two matrix multiplications. To compute the $|B'_{a,d}|$ values, multiply the two matrices M, M' which have entries from $\{0, 1\}$, with $M_{a,b} = 1$ iff $\{a, b\} \in E(G) \cap A \times B$ and $M'_{b,d} = 1$ iff $\{b, d\} \notin E(G) \cap B \times D$. Observe that

$|B'_{a,d}|$ is exactly $(M \cdot M')_{a,d}$. To compute $|C'_{a,d}|$, multiply M, M' over $\{0,1\}$ where $M_{a,c} = 1$ iff $\{a,c\} \notin E(G) \cap A \times C$ and $M'_{c,d} = 1$ iff $\{c,d\} \in E(G) \cap C \times D$.

After having these estimates and computing APMVC on H, it can be decided whether G contains a 4-clique in $O(n^2)$ time as follows. Go through all edges $\{a,d\} \in E(G) \cap A \times D$ and decide whether the edge participates in a 4-clique by comparing $|B'_{a,d}| + |C'_{a,d}|$ to the node connectivity $NC(a,d)$ in H. By the above claim, an edge $\{a,d\}$ with $NC(a,d) > |B'_{a,d}| + |C'_{a,d}|$ is found if and only if there is a 4-clique in G. The total running time is $O(T(n) + MM(n))$, which completes the proof of Lemma 3.4.2. □

3.4.2 Reduction to the k-Bounded Case

Next, we exploit a certain versatility of the reduction and adapt it to ask only about min-cut values (aka node connectivities) that are smaller than k. In other words, we will reduce to the k-bounded version of All-Pairs Min-Cut with unit node-capacities (abbreviated kAPMVC, for k-bounded All-Pairs Minimum Vertex-Cut). Our lower bound improves on the $\Omega(n^\omega)$ conjectured lower bound for Transitive Closure as long as $k = \omega(n^{1/2})$.

Lemma 3.4.4. *Suppose kAPMVC on n-node DAGs with unit node-capacities can be solved in time $T(n,k)$. Then 4-Clique on n-node graphs can be solved in time $O(\frac{n^2}{k^2} \cdot T(n,k) + MM(n))$, where $MM(n,n)$ is the time to multiply two matrices from $\{0,1\}^{n \times n}$.*

Proof of Lemma 3.4.4. Given a 4-partite graph G as in the definition of the 4-Clique problem, $O(n^2/k^2)$ graphs H are constructed in a way that is similar to the previous reduction, and an algorithm for kAPMVC is called on each of these graphs. Assume w.l.o.g. that k divides n and partition the sets A, D arbitrarily to sets $A_1, \ldots, A_{n/k}$ and $D_1, \ldots, D_{n/k}$ of size k each. For each pair of integers $i, j \in [n/k]$, generate one graph H_{ij} by restricting the attention to the nodes of G in A_i, B, C, D_j and looking for a 4-clique only there.

Let us fix a pair $i, j \in [n/k]$ and describe the construction of H_{ij}. To simplify the description, let us omit the subscripts i, j, referring to this graph as H, and think of G as having four parts A, B, C, D, where A and D are in fact A_i, D_j and are therefore smaller: $|A| = |D| = k$ and $|B| = |C| = n$.

The nodes in H are partitioned into four sets A', B, C, D', where the sets B, C are the same as in G. For the nodes in A, D in G, multiple copies are created in H. For all integers $x \in [n/k]$ and node $a \in A$ in G, add a node a_x to A' in H. Similarly, for all $x \in [n/k]$ and node $d \in D$, add a node a_x to A'. Note that H contains $O(n)$ nodes.

To define the edges, partition the nodes in B and C arbitrarily to sets $B_1, \ldots, B_{n/k}$ and $C_1, \ldots, C_{n/k}$ of size k. Now, the edges are defined in a similar way to the previous proof, except each a_x is connected only to nodes in B_x, and each d_y is connected only to nodes in C_y. More formally:

- (A to B) For every $a_x \in A', b \in B_x$ such that $\{a,b\} \in E(G)$, add to $E(H)$ a directed edge (a_x, b).
- (B to C) For every $b \in B, c \in C$ such that $\{b,c\} \in E(G)$, add to $E(H)$ a directed edge (b,c).
- (C to D) For every $c \in C_y, d_y \in D'$ such that $\{c,d\} \in E(G)$, add to $E(H)$ a directed edge (c, d_y).

- (A to C) For every $a_x \in A', c \in C$ such that $\{a, c\} \notin E(G)$, add to $E(H)$ a directed edge (a_x, c).
- (B to D) For every $b \in B, d_y \in D'$ such that $\{b, d\} \notin E(G)$, add to $E(H)$ a directed edge (b, d_y).

This completes the construction of H. The arguments for correctness follow the same lines as in the previous proof. For nodes $a_x \in A', d_y \in D'$ denote:

$$B'_{a_x,d_y} = \{b \in B_x \mid \{a,b\} \in E(G) \text{ and } \{b,d\} \notin E(G)\},$$
$$C'_{a_x,d_y} = \{c \in C_y \mid \{a,c\} \notin E(G) \text{ and } \{c,d\} \in E(G)\}.$$

Claim 3.4.5. *Let $a_x \in A', d_y \in D'$ be nodes with $\{a, d\} \in E(G)$. If the edge $\{a, d\}$ does not participate in a 4-clique in G together with any nodes in $B_x \cup C_y$, then the node connectivity from a_x to d_y in H, denoted $NC(a_x, d_y)$, is exactly*

$$NC(a_x, d_y) = |B'_{a_x,d_y}| + |C'_{a_x,d_y}|$$

and otherwise $NC(a_x, d_y)$ is strictly larger.

Proof of Claim 3.4.5. The proof is very similar to the one in the previous reduction.

We start by observing that all paths from a_x to d_y in H can have either two or three hops.

For the first direction, assuming that there is a 4-clique (a, b^*, c^*, d) in G with $b^* \in B_x, c^* \in C_y$, we show a set P of node-disjoint paths from a_x to d_y of size $|B'_{a_x,d_y}| + |C'_{a_x,d_y}| + 1$. For all nodes $b \in B'_{a_x,d_y}$, add the 2-hop path $a_x \to b \to d_y$ to P. For all nodes $c \in C'_{a_x,d_y}$, add the 2-hop path $a_x \to c \to d_y$ to P. So far, all these paths are clearly node-disjoint. Then, add the 3-hop path $a_x \to b^* \to c^* \to d_y$ to P. This path is node-disjoint from the rest because $b^* \notin B'_{a_x,d_y}$ (because $\{b^*, d\} \in E(G)$) and $c^* \notin C'_{a_x,d_y}$ (because $\{a, c^*\} \in E(G)$).

For the second direction, assume that there do not exist nodes $b \in B_x, c \in C_y$ that complete a 4-clique with a, d. In this case, for every set P of node-disjoint paths from a_x to d_y, there is a set P' of 2-hop node-disjoint paths from a_x to d_y that has the same size. To see this, let $a_x \to b \to c \to d_y$ be some 3-hop path in P. Since (a, b, c, d) is not a 4-clique in G and $\{a, d\}, \{a, b\}, \{b, c\}, \{c, d\}$ are edges in G, it follows that either $\{a, c\} \notin E(G)$ or $\{b, d\} \notin E(G)$. If $\{a, c\} \notin E(G)$ then $a_x \to c$ is an edge in H and the 3-hop path can be replaced with the 2-hop path $a_x \to c \to d_y$ (by skipping b) and one is remained with a set of node-disjoint paths of the same size. Similarly, if $\{b, d\} \notin E(G)$ then $b \to d_y$ is an edge in H and the 3-hop path can be replaced with the 2-hop path $a_x \to b \to d_y$. This can be done for all 3-hop paths and result in P'. Finally, note that the number of 2-hop paths from a_x to d_y is exactly $|B'_{a_x,d_y}| + |C'_{a_x,d_y}|$, and this completes the proof of Claim 3.4.5. $\square$

This claim implies that in order to determine whether a pair $a \in A, d \in D$ participate in a 4-clique in G is it is enough to check whether $\sum_{x,y \in [n/k]} NC(a_x, d_y)$ is equal to $\sum_{x,y \in [n/k]} |B'_{a_x,d_y}| + |C'_{a_x,d_y}|$. Note that the latter is equal to $|B'_{a,d}| + |C'_{a,d}|$ according to the notation in the previous reduction:

$$B'_{a,d} = \{b \in B \mid \{a,b\} \in E(G) \text{ and } \{b,d\} \notin E(G)\},$$
$$C'_{a,d} = \{c \in C \mid \{a,c\} \notin E(G) \text{ and } \{c,d\} \in E(G)\}.$$

Computing the estimates To complete the reduction, observe that the values $|B'_{a,d}|+|C'_{a,d}|$ can be computed for all pairs $a \in A, d \in D$ (for all sub-instances i,j) using two matrix products, just like in the previous reduction.

After having these estimates and computing APMVC on H, it can be decided whether G contains a 4-clique in $O(k^2 \cdot n/k)$ time, for each sub-instance i,j, as follows. Go through all edges $\{a,d\} \in E(G) \cap A_i \times D_j$ and check whether the edge participates in a 4-clique by comparing this value $|B'_{a,d}|+|C'_{a,d}|$ to the node connectivities $\sum_{x,y\in[n/k]} NC(a_x,d_y)$ in H. By the above claim, one can find an edge $\{a,d\}$ with $\sum_{x,y\in[n/k]} NC(a_x,d_y) > |B'_{a,d}|+|C'_{a,d}|$ if and only if there is a 4-clique in G. The total running time is $O(\frac{n^2}{k^2}\cdot T(n,k)+MM(n))$, which completes the proof of Lemma 3.4.4. □

Proof of Theorem 3.1.1. Assume there is an algorithm that solves kAPMVC in time $O((n^{\omega-1}k^2)^{1-\varepsilon})$. Then by Lemma 3.4.4 there is an algorithm that solves 4-Clique in time $= O(\frac{n^2}{k^2}\cdot(n^{\omega-1}k^2)^{1-\varepsilon} + MM(n)) \leq O(n^{\omega+1-\varepsilon'})$, for some $\varepsilon' > 0$. The bound for combinatorial algorithms is achieved similarly. □

3.5 Randomized Algorithms for General Digraphs

In this section we develop faster randomized algorithms for the following problems. Given a digraph $G=(V,E)$ with unit vertex-capacities, two subsets $S,T \subseteq V$ and parameter $k>0$, find all $s \in S, t \in T$ for which the minimum st-cut value is less than k and report their min-cut value. This problem is called kSTMVC, and if $S=T=V$, it is called kAPMVC. This is done by showing that the framework of Cheung et al. [CLL13] can be applied faster to unit vertex-capacitated graphs. Before providing our new algorithmic results (in Theorem 3.5.2 and Corollary 3.5.3), we first give some background on network coding (see [CLL13] for a more comprehensive treatment).

The Network-Coding Approach. Network coding is a novel method for transmitting information in a network. As shown in a fundamental result [ACLY00], if the edge connectivity from the source s to each sink t_i is $\geq k$, then k units of information can be shipped to all sinks simultaneously by performing encoding and decoding at the vertices. This can be seen as a max-information-flow min-cut theorem for multicasting, for which an elegant algebraic framework has been developed for constructing efficient network coding schemes [LYC03, KM03]. These techniques were used in [CLL13] to compute edge connectivities, and below we briefly recap their method and notation.

Given a vertex s from which we need to compute the maximum flow to all other vertices in G, define the following matrices over a field $\mathbb{F}$.

- $F_{d\times m}$ is a matrix whose m columns are d-dimensional global encoding vectors of the edges, with $d=\deg_G^{out}(s)$.
- $K_{m\times m}$ is a matrix whose entry (e_1,e_2) corresponds to the local encoding coefficient k_{e_1,e_2} which is set to a random value from the field $|\mathbb{F}|=O(m^c)$ if e_1's head is e_2's tail, and to zero otherwise.
- $H_{d\times m}$ is a matrix whose columns are $(\overrightarrow{e_1},\ldots,\overrightarrow{e_d},\overrightarrow{0},\ldots,\overrightarrow{0})$ ($\overrightarrow{e_i}$ is in the column corresponding to e_i) where the column vector $\overrightarrow{e_i}$ is the ith standard basis vector and $e_1,\ldots,e_d$ are the edges outgoing of s.

The global encoding vectors F are defined such that $F = FK + H$, and then by simple manipulations the equation $F = H(I-K)^{-1}$ is achieved (so multiplying by H simply picks rows of $(I-K)^{-1}$ that correspond to the edges outgoing of s). The algorithm utilizes this by first computing $(I-K)^{-1}$ in time $O(m^{\omega})$, and then for every source s and sink t computing the rank of the submatrix corresponding to rows $\delta^{out}(s)$ and columns $\delta^{in}(t)$ in time $O(m^2 n^{\omega-2})$, and the overall time is $O(m^{\omega})$ since $m \geq n$. Notice that even if we only care about the maximum flow between given sets of sources and targets S, T, the bottleneck is that the matrix $(I-K)$ has m^2 entries, potentially most are non-zeroes, which must be read to compute $(I-K)^{-1}$.

Our Algorithmic Results.

Lemma 3.5.1. *kSTMVC can be solved in randomized time* $O\Big(n^{\omega} + \sum_{s\in S}\sum_{t\in T} \deg_G^{out}(s)\deg_G^{in}(t)^{\omega-1}\Big)$, *where* $\deg_G^{out}(u)$ *and* $\deg_G^{in}(u)$ *denote the out-degree and the in-degree, respectively, of vertex* u *in the input graph* G.

Proof. We consider global encoding vectors in the vertices rather than in the edges in the natural way, namely, the coefficients are non-zero for every pair of adjacent vertices (rather than adjacent edges), and for a source s and a sink t we compute the rank of the submatrix of $(I-K)^{-1}$ whose rows correspond to the vertices $N^{out}(s)$ and columns correspond to $N^{in}(t)$. The running time is dominated by inverting the matrix $(I-K)_{n\times n}$ and computing the rank of the relevant submatrices, that is $O(n^{\omega} + \sum_{s\in S}\sum_{t\in T}\deg_G(s)\deg_G(t)^{\omega-1})$, as required. Notice that by considering vertices rather than edges, the bottleneck moves from computing $(I-K)^{-1}$ to computing the rank of the relevant submatrices.

To prove the correctness, we argue that Theorem 2.1 from [CLL13] holds also here (adjusted to node capacities). Part 1 in their proof clearly holds also here, so we focus on the second part, which in [CLL13] shows that the edge connectivity from s to t, denoted $\lambda_{s,t}$, is equal to the rank of the matrix $M_{s,t}$ of size $\deg_G(s) \times \deg_G(t)$ comprising of the global encoding vectors on the edges incoming to t as its columns. Here, we denote the vertex connectivity from s to t by $\kappa_{s,t}$, and the corresponding matrix $M_{s,t}^{vertices}$, and we show that their proof can be adjusted to show $\text{rank}(M_{s,t}^{vertices}) = \kappa_{s,t}$, as required. First, $\text{rank}(M_{s,t}^{vertices}) \leq \kappa_{s,t}$ as instead of considering an edge-cut (S,T) and claiming that the global encoding vector on each incoming edge of t is a linear combination of the global encoding vectors of the edges in (S,T), we consider a node-cut $(S^{vertices}, C^{vertices}, T^{vertices})$, and similarly claim that the global encoding vectors on each vertex with an edge to t is a linear combination of the global encoding vectors in $C^{vertices}$, and the rest of the proof follows. For the second part, we argue that $\text{rank}(M_{s,t}^{vertices}) \geq \kappa_{s,t}$. The main proof idea from [CLL13] that the rank does not increase if we restrict our attention to a subgraph holds here too, only that we use vertex disjoint paths as the subgraph to establish the rank. □

Theorem 3.5.2. *kSTMVC can be solved in randomized time* $O\Big(\big(n + ((|S|+|T|)k)\big)^{\omega} + |S||T|k^{\omega}\Big)$.

Proof. In order to use Lemma 3.5.1 to prove Theorem 3.5.2, we need to decrease the degree of sources S and sinks T. Thus, for every source s we add a layer of k vertices L_s and connect s to all the vertices in L_s which in turn are connected by a complete directed bipartite graph to the set of vertices $N^{out}(s)$, directed away from L_s. Similarly, for every sink $t \in T$ we add a layer of k vertices L_t and connect to t all the vertices in L_t, which in turn are connected by a complete directed bipartite graph from the set of vertices $N^{in}(t)$, directed away from $N^{in}(t)$. Note that all flows of size $\leq k-1$ are preserved, and flows of size $\geq k$ become k. This incurs an additive term $(|S|+|T|)k$ in the dimension of the matrix inverted, and altogether we achieve a running time of $O((n+(|S|+|T|)k)^{\omega}+|S||T|k^{\omega})$, as required. $\square$

As an immediate corollary we have the following.

Corollary 3.5.3. *kAPMVC can be solved in randomized time $O((nk)^{\omega})$.*

Chapter 4

New Algorithms and Lower Bounds for All-Pairs Max-Flow in Unit-Capacity Undirected Graphs[1]

4.1 Introduction

In the maximum *st*-flow problem (abbreviated Max-Flow), the goal is to compute the maximum value of a feasible flow between a given pair of nodes s, t (sometimes called *terminals*) in an input graph. [2] Determining the time complexity of this problem is one of the most prominent open questions in fine-grained complexity and algorithms. The best running time known for directed (or undirected) graphs with n nodes, m edges, and largest integer capacity U is $\tilde{O}(\min\{m^{10/7}U^{1/7},\ m^{11/8}U^{1/4}, m^{4/3}U^{1/3}, m\sqrt{n}\log U\})$ [Mad16, LS20a, LS20b, LS14], where throughout $\tilde{O}(f)$ hides logarithmic factors and stands for $O(f\log^{O(1)} f)$. To date, there is no $\Omega(m^{1+\varepsilon})$ lower bound for this problem, even when utilizing one of the popular conjectures of fine-grained complexity, such as the Strong Exponential-Time Hypothesis (SETH) of [IP01]. [3] This gap is regularly debated among experts, and a common belief is that such a lower bound is not possible, since a near-linear-time algorithm exists but is not yet known. There is also a formal barrier for basing a lower bound for Max-Flow on SETH, as it would refute the so-called Non-deterministic SETH (NSETH) [CGI+16]. We will henceforth assume that Max-Flow can be solved in time $m^{1+o(1)}$, and investigate some of the most important questions that remain open under this favorable assumption. (None of our results need this assumption; it only serves for highlighting their significance.)

Perhaps the most natural next-step after the s, t version is the "all-pairs" version (abbreviated All-Pairs Max-Flow), where the goal is to solve Max-Flow for all pairs of nodes in the graph. This multi-terminal problem, dating back to 1960 [May60, Chi60], is the main focus of our work:

What is the time complexity of computing Max-Flow between all pairs of nodes?

[1] This chapter is based on [AKT20b].

[2] Throughout, we focus on computing the value of the flow (rather than an actual flow), which is equal to the value of the minimum *st*-cut by the famous max-flow/min-cut theorem [FF56].

[3] SETH asserts that for every fixed $\varepsilon > 0$ there is an integer $k \geq 3$, such that kSAT on n variables and m clauses cannot be solved in time $2^{(1-\varepsilon)n}m^{O(1)}$.

We will discuss a few natural settings, e.g., directed vs. undirected, or node capacities vs. edge capacities, in which the answer to this question may vary. A trivial strategy for solving this problem (in any setting) is to invoke a $T(m)$-time algorithm for the s,t version $O(n^2)$ times, giving a total time bound of $O(n^2) \cdot T(m)$, which is $n^2 \cdot m^{1+o(1)}$ under our favorable assumption. But one would hope to do much better, as this all-pairs version arises in countless applications, such as a graph-clustering approach for image segmentation [WL93].

In undirected edge-capacitated graphs, a seminal paper of Gomory and Hu [GH61] showed in 1961 how to solve All-Pairs Max-Flow using only $n-1$ calls to a Max-Flow algorithm, rather than $O(n^2)$ calls, yielding an upper bound $O(n) \cdot T(m)$. (See also [Gus90] for a different algorithm where all the $n-1$ calls can be executed on the original graph.) This time bound has improved over the years, following the improvements in algorithms for Max-Flow, and under our assumption it would ultimately be $n \cdot m^{1+o(1)}$. Even more surprisingly, Gomory and Hu showed that all the n^2 answers can be represented using a single tree, which can be constructed in the same time bound. Formally, A *cut-equivalent tree* to a graph G is an edge-capacitated tree T on the same set of nodes, with the property that for every pair of nodes s,t, every minimum st-cut in T yields a bipartition of the nodes which is a minimum st-cut in G, and of the same value as in T. [4] See also [GT01] for an experimental study, and the Encyclopedia of Algorithms [Pan16] for more background. The only algorithm that constructs a cut-equivalent tree without making $\Omega(n)$ calls to a Max-Flow algorithm was designed by Bhalgat, Hariharan, Kavitha, and Panigrahi [BHKP07]. It runs in time $\tilde{O}(mn)$ in unit-capacity graphs (or equivalently, if all edges have the same capacity), and utilizes a tree-packing approach that was developed in [CH03, HKP07], inspired by classical results of [Gab95] and [Edm70]. However, if Max-Flow can indeed be computed in near-linear time, then none of the later algorithms beat by a polynomial factor the time bound $n \cdot m^{1+o(1)}$ of Gomory and Hu's half-century old algorithm.

The time complexity of All-Pairs Max-Flow becomes higher in settings where Gomory and Hu's "tree structure" [GH61] does not hold. For instance, in node-capacitated graphs (where the flow is constrained at intermediate nodes, [5] rather than edges) flow-equivalent trees are impossible, since there could actually exist $\Omega(n^2)$ different maximum-flow values in a single graph [HL07] (see therein also an interesting exposition of certain false claims made earlier). Directed edges make the all-pairs problem even harder; in fact, in this case node capacities and edge capacities are equivalent, and thus this setting does not admit flow-equivalent trees, see [May62, Jel63, HL07]. In the last decade, different algorithms were proposed to beat the trivial $O(n^2) \cdot T(m)$ time bound in these harder cases. The known bound for general graphs is $O(m^\omega)$, due to Cheung, Lau, and Leung [CLL13], where $\omega < 2.38$ is the matrix multiplication exponent. A related version, which is obviously no harder than All-Pairs Max-Flow, is to ask (among all pairs of nodes) only for flow values that are at most k, assuming unit node-capacities; for example, the case $k = 1$ is the transitive closure problem (reachability). For $k = 2$, an $\tilde{O}(n^\omega)$-time algorithm was shown in [GGI$^+$17], and very recently a similar bound was

[4]Notice that a minimum st-cut in T consists of a single edge that has minimum capacity along the unique st-path in T, and removing this edge disconnects T to two connected components. A *flow-equivalent tree* has the weaker property that for every pair of nodes s,t, the maximum st-flow value in T equals that in G. The key difference is that flow-equivalence maintains only the *values* of the flows (and thus also of the corresponding cuts).

[5]Granot and Hassin [GH86] considered a related but different notion of minimum st-cuts with node capacities, where the flow is constrained also by the capacities of the source and sink (in addition to intermediate nodes), and so an equivalent tree exists and can be computed. This makes the problem much easier.

achieved for all $k = O(1)$ [AGI+19]. The aforementioned papers [CLL13, GGI+17, AGI+19] also present improved algorithms for acyclic graphs (DAGs). In addition, essentially optimal $\tilde{O}(n^2)$-time algorithms were found for All-Pairs Max-Flow in certain graph families, including small treewidth [ACZ98], planar graphs [LNSW12], and surface-embedded graphs [BENW16].

The framework of fine-grained complexity has been applied to the all-pairs problem in a few recent papers, although its success has been limited to the directed case. Abboud, Vassilevska-Williams, and Yu [AWY18] proved SETH-based lower bounds for some multi-terminal variants of Max-Flow, such as the single-source all-sinks version, but not all-pairs. Krauthgamer and Trabelsi [KT18b] proved that All-Pairs Max-Flow cannot be solved in time $O(n^{3-\varepsilon})$, for any fixed $\varepsilon > 0$, unless SETH is false, even in the sparse regime $m = n^{1+o(1)}$. This holds also for unit-capacity graphs, and it essentially settles the complexity of the problem for directed sparse graphs, showing that the $O(n^2) \cdot T(m)$ upper bound is optimal if one assumes that $T(m) = m^{1+o(1)}$. Recently, Abboud et al. [AGI+19] proved a conditional lower bound that is even higher for dense graphs, showing that an $O(n^{\omega+1-\varepsilon})$-time algorithm would refute the 4-Clique conjecture. However, no non-trivial lower bound is known for undirected graphs.

4.1.1 The Challenge of Lower Bounds in Undirected Graphs

Let us briefly explain the difficulty in obtaining lower bounds for undirected graphs. Consider the following folklore reduction from Boolean Matrix Multiplication (BMM) to All-Pairs Reachability in directed graphs (the aforementioned special case of All-Pairs Max-Flow with $k = 1$). In BMM the input is two $n \times n$ boolean matrices P and Q, and the goal is to compute the product matrix R given by

$$R(a, c) := \vee_{b=1}^{n} (P(a, b) \wedge Q(b, c)), \qquad \forall a, c \in [n].$$

Computing R can be reduced to All-Pairs Reachability as follows. Construct a graph with three layers A, B, C with n nodes each, where the edges are directed $A \to B \to C$ and represent the two matrices: $a \in A$ is connected to $b \in B$ iff $P(a, b) = 1$; and $b \in B$ is connected to $c \in C$ iff $Q(b, c) = 1$. It is easy to see that $R(a, c) = 1$ iff node $a \in A$ can reach node $c \in C$ (via a two-hop path).

This simple reduction shows an $n^{\omega-o(1)}$ lower bound for All-Pairs Reachability in dense directed graphs assuming the BMM conjecture (see [AW14]), which states that any combinatorial algorithm (i.e. one that does not use fast matrix multiplication techniques) that solves BMM requires $n^{3-o(1)}$ time. Higher lower bounds can be proved by more involved reductions that utilize the extra power of flow over reachability, e.g., an $n^{3-o(1)}$ lower bound in sparse directed graphs assuming SETH [KT18b]. Nevertheless, this simple reduction illustrates the main difficulty in adapting such reductions to undirected graphs.

Consider the same construction but with *undirected edges* (i.e., without the edge orientations). The main issue is that paths from A to C can now have more than two hops – they can crisscross between two adjacent layers before moving on to the next one. Indeed, it is easy to construct examples in which the product $R(a, c) = 0$ but there is a path from a to c (with more than two hops). Even if we try to use the extra power of flow, giving us information about the number of paths rather than just the existence of a path, it is still unclear how to distinguish flow that uses a two-hop path (YES case) from flow that uses only longer paths (NO case).

A main technical novelty of this work is a trick to overcome this issue. The high-level idea is to design large gaps between the capacities of nodes in different layers in order to incentivize flow to move to the "next layer". Let us exhibit how this trick applies to the simple reduction above. Remove the edge orientations from our three-layer graph, and introduce node capacities, letting all nodes in B, the middle layer, have capacity $2n$, and all nodes in $A \cup C$, the other two layers, have capacity 1. Now, consider the maximum flow from $a \in A$ to $c \in C$. If $R(a,c) = 1$ then there is a two-hop path through some $b \in B$, which can carry $2n$ units of flow, hence the maximum-flow value is at least $2n$. On the other hand, if $R(a,c) = 0$ then every path from a to c must have at least four hops, and a maximum flow must be composed of such paths. Any such path must pass through at least one node in $A \cup C \setminus \{a,c\}$, whose capacity is only 1, hence the maximum flow is bounded by $|A \cup C \setminus \{a,c\}| = 2n - 2$. This proves the same $n^{\omega - o(1)}$ lower bound as before, but now for undirected graphs with node capacities. We note that the argument for undirected graphs can be simplified a bit if we allow nodes of capacity 0. In Section 4.4 we utilize this trick in a more elaborate way to prove stronger lower bounds.

4.1.2 Our Results

Our main negative result is the first (conditional) lower bound for All-Pairs Max-Flow that holds in undirected graphs. For sparse, node-capacitated graphs we are able to match the lower bound $n^{3-o(1)}$ that was previously known only for directed graphs [KT18b], and it also matches the hypothetical upper bound $n^{3+o(1)}$.

Theorem 4.1.1. *Assuming SETH, no algorithm can solve All-Pairs Max-Flow in undirected graphs on n nodes and $O(n)$ edges with node capacities in $[n^2]$ in time $O(n^{3-\varepsilon})$ for some fixed $\varepsilon > 0$.*

Our lower bound holds even under assumptions that are weaker than SETH (see Section 4.4), as we reduce from the 3-Orthogonal-Vectors (3OV) problem. At a high level, it combines the trick described above for overcoming the challenge in undirected graphs, with the previous reduction of [KT18b] from 3OV to the directed case. However, both of these ingredients have their own subtleties and fitting them together requires adapting and tweaking them very carefully.

Following our Theorem 4.1.1, the largest remaining gap in our understanding of All-Pairs Max-Flow concerns the most basic and fundamental setting: undirected graphs with edge capacities. What is the time complexity of computing a cut-equivalent tree? The upper bound has essentially been stuck at $n \cdot m^{1+o(1)}$ for more than half a century, while we cannot even rule out a near-linear $m^{1+o(1)}$ running time. To our great surprise, after a series of failed attempts at proving any lower bound, we have noticed a simple way to design a new algorithm for computing cut-equivalent trees for graphs with unit capacities, breaking the longstanding mn barrier!

Theorem 4.1.2. *There is an algorithm that, given an undirected graph G with n nodes and m edges (and unit edge-capacities) and parameter $1 \le d \le n$, constructs a cut-equivalent tree in time $\tilde{O}(md + \Phi(m,n,d))$, where $\Phi(m,n,d) = \max\Big\{ \sum_{i=1}^{m/d} T(m,n,F_i) \ : \ F_1, \dots, F_{m/d} \ge 0, \sum_{i=1}^{m/d} F_i \le 2m \Big\}$ and $T(m,n,F)$ is the time bound for Max-Flow on instances where whose flow value is at most a F.*

Using the current bound on $T(m,n,F)$ we achieve running time $\tilde{O}(m^{3/2}n^{1/6})$, and under the plausible hypothesis that $T(m,n) = m^{1+o(1)}$ our time bound becomes $m^{3/2+o(1)}$. In the regime of sparse graphs where $m = \tilde{O}(n)$ the previous best algorithm of Bhalgat et al. [BHKP07] had running time $\tilde{O}(n^2)$, whereas we achieve $\tilde{O}(n^{5/3})$, or conditionally $n^{3/2+o(1)}$. In fact, we improve on their upper bound as long as $m = O(n^{5/3-\varepsilon})$. Clearly, this also leads to improved bounds for All-Pairs Max-Flow (with unit edge-capacities), for which the best strategy known is to compute the tree and then extract the answers in time $O(n^2)$.

The main open question remains: Can we prove any super-linear lower bounds for the edge-capacitated case in undirected graphs? Is there an $m^{1+\varepsilon}$ lower bound under SETH for constructing a cut-equivalent tree? Perhaps surprisingly, we prove a strong barrier for the possibility of such a result.

We follow the non-reducibility framework of Carmosino et al. [CGI$^+$16]. Intuitively, if problem A is conjectured to remain hard for nondeterministic algorithms while problem B is known to become significantly easier for such algorithms, then we should not expect a reduction from A to B to exist. Such a reduction would allow the nondeterministic speedups for problem B to carry over to A. To formalize this connection, Carmosino et al. introduce NSETH: the hypothesis that SETH holds against co-nondeterministic algorithms. NSETH is plausible because it is not clear how a powerful prover could convince a sub-2^n-time verifier that a given CNF formula is *not* satisfiable. Moreover, it is known that refuting NSETH requires new techniques since it implies new circuit lower bounds. Then, Carmosino et al. exhibited nondeterministic (and co-nondeterministic) speedups for problems such as 3-SUM and Max-Flow (using LP duality), showing that a reduction from SAT to these problems would refute NSETH.

Our final result builds on Theorem 4.1.2 to design a near-linear time [6] *nondeterministic* algorithm for constructing a cut-equivalent tree. This algorithm can perform nondeterministic choices and in the end, outputs either a correct cut-equivalent tree or "don't know" (i.e., aborts), however we are guaranteed that for every input graph there is at least one sequence of nondeterministic choices that leads to a correct output. Our result could have applications in computation-delegation settings and may be of interest in other contexts. In particular, since our nondeterministic witness can be constructed deterministically efficiently, namely, in polynomial but super-linear time, it provides a potentially interesting *certifying algorithm* [MMNS11, ABMR11] (see [Kün18] for a recent paper with a further discussion of the connections to fine-grained complexity). Our final non-reducibility result is as follows.

Theorem 4.1.3. *If for some $\varepsilon > 0$ there is a deterministic fine-grained reduction proving an $\Omega(m^{1+\varepsilon})$ lower bound under SETH for constructing a cut-equivalent tree of an undirected unit edge-capacitated graph on m edges, then NSETH is false.*

Our result (and this framework for non-reducibility) does not address the possibility of proving a SETH based lower bound with a randomized fine-grained reduction. This is because NSETH does not remain plausible when faced against randomization (see [CGI$^+$16, Wil16]). That said, we are not aware of any examples where this barrier has been successfully bypassed with randomization.

Roadmap. Our main algorithm is described in the Section 4.2. The nondeterministic algorithm and non-reducibility result are presented in Section 4.3. We then present our lower bounds in Section 4.4. The last section discusses open questions.

[6] We say that a time bound $T(n)$ is near-linear if it is bounded by $O(n \log^c n)$ for some constant $c > 0$.

4.2 Algorithm for a Cut-Equivalent Tree

The basic strategy in our algorithm for unit edge-capacities is to handle separately nodes whose connectivity (to other nodes) is high from those whose connectivity is low. The motivation comes from the simple observation that the degree of a node is an upper bound on the maximum flow from this node to any other node in the graph. Specifically, our algorithm has two stages. The first stage uses one method (of partial trees [HKP07, BHKP07]), to compute the parts of the tree that correspond to small connectivities, and the second stage uses another method (the classical Gomory-Hu algorithm [GH61]) to complete it to a cut-equivalent tree (see Figure 4.1). Let us briefly review these two methods.

The Gomory-Hu algorithm. This algorithm constructs a cut-equivalent tree $\mathcal{T}$ in iterations. Initially, $\mathcal{T}$ is a single node associated with V (the node set of G), and the execution maintains the invariant that $\mathcal{T}$ is a tree; each tree node i is a *super-node*, which means that it is associated with a subset $V_i \subseteq V$; and these super-nodes form a partition $V = V_1 \sqcup \cdots \sqcup V_l$. At each iteration, the algorithm picks arbitrarily two graph nodes s, t that lie in the same tree super-node i, i.e., $s, t \in V_i$. The algorithm then constructs from G an auxiliary graph G' by merging nodes that lie in the same connected component of $\mathcal{T} \setminus \{i\}$ and invokes a $\mathsf{Max\text{-}Flow}$ algorithm to compute in this G' a minimum st-cut, denoted C'. (For example, if the current tree is a path on super-nodes $1, \ldots, l$, then G' is obtained from G by merging $V_1 \cup \cdots \cup V_{i-1}$ into one node and $V_{i+1} \cup \cdots \cup V_l$ into another node.) The submodularity of cuts ensures that this cut is also a minimum st-cut in the original graph G, and it clearly induces a partition $V_i = S \sqcup T$ with $s \in S$ and $t \in T$. The algorithm then modifies $\mathcal{T}$ by splitting super-node i into two super-nodes, one associated with S and one with T, that are connected by an edge whose weight is the value of the cut C', and further connecting each neighbor of i in $\mathcal{T}$ to either S or T (viewed as super-nodes), depending on its side in the minimum st-cut C' (more precisely, neighbor j is connected to the side containing V_j).

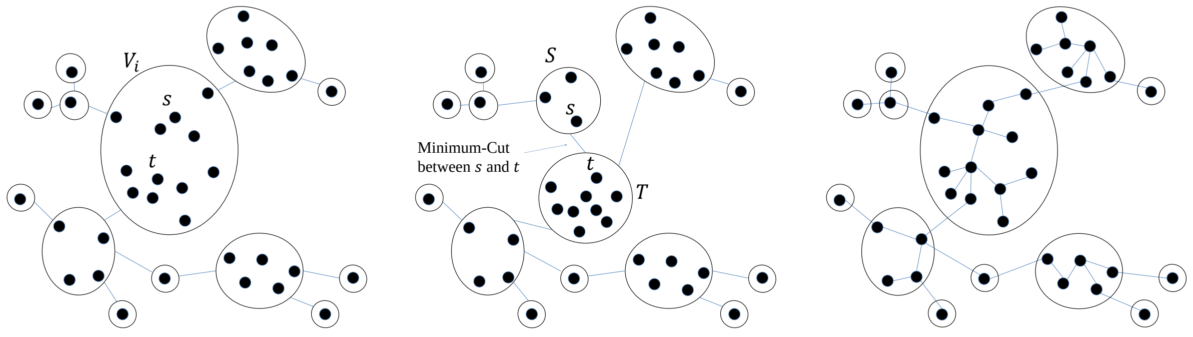

Figure 4.1: An illustration of the construction of $\mathcal{T}$. Left: $\mathcal{T}$ right before the partition of the super-node V_i. Middle: after the partitioning of V_i Right: $\mathcal{T}$ as it unfolds after the Gomory-Hu algorithm finishes.

The algorithm performs these iterations until all super-nodes are singletons, and then $\mathcal{T}$ is a weighted tree with effectively the same node set as G. It can be shown [GH61] that for every $s, t \in V$, the minimum st-cut in $\mathcal{T}$, viewed as a bipartition of V, is also a minimum st-cut in G, and of the same cut value. We stress that this property holds regardless of the choice made at each step of two nodes $s \neq t \in V_i$.

Partial Tree. A k-partial tree, formally defined below, can also be thought of as the result of contracting all edges of weight greater than k in a cut-equivalent tree of G. Such a tree can obviously be constructed using the Gomory-Hu algorithm, but as stated below (in Lemma 4.2.2), faster algorithms were designed in [HKP07, BHKP07], see also [Pan16, Theorem 3]. We show below (in Lemma 4.2.3) that such a tree can be obtained also by a truncated execution of the Gomory-Hu algorithm, and finally we use this simple but crucial fact to prove our main theorem.

Definition 4.2.1 (k-Partial Tree [HKP07])**.** *A* k-partial tree *of a graph $G = (V, E)$ is a weighted tree on $l \leq |V|$ super-nodes constituting a partition $V = V_1 \sqcup \cdots \sqcup V_l$, with the following property: For every two nodes $s, t \in V$ whose minimum-cut value in G is at most k, s and t lie in different super-nodes $s \in S$ and $t \in T$, such that the minimum ST-cut in the tree defines a bipartition of V which is a minimum st-cut in G and has the same value.*

Lemma 4.2.2 ([BHKP07])**.** *There is an algorithm that given an undirected graph with n nodes and m edges with unit edge-capacities and an integer $k \in [n]$, constructs a k-partial tree in time $\tilde{O}(mk)$.*

Lemma 4.2.3. *Given a k-partial tree T_{low} of a graph $G = (V, E)$, there is a truncated execution of the Gomory-Hu algorithm that produces T_{low} (i.e., its auxiliary tree $\mathcal{T}$ becomes T_{low}).*

Proof. Consider an execution of the Gomory-Hu algorithm with the following choices. At each iteration, pick any two nodes $s, t \in V$ that lie in the same super-node i of the current tree $\mathcal{T}$ (hence they are a feasible choice in a Gomory-Hu execution) *but furthermore* lie in different super-nodes of T_{low}, as long as such s, t exist. Then split super-node i of $\mathcal{T}$ using the minimum st-cut induced by T_{low} (rather than an arbitrary minimum st-cut). As this cut corresponds to an edge in T_{low}, it cannot split any super-node of T_{low}, which implies, by an inductive argument, that the super-nodes of T_{low} are subsets of the super-nodes of $\mathcal{T}$, and thus our chosen cut is a feasible choice for a Gomory-Hu execution. In order to claim that $\mathcal{T}$ will have the same tree structure as T_{low}, we also use the following simple inductive argument. Each pair A, B of super-nodes of $\mathcal{T}$ connected with an edge of capacity c has a pair $A' \subseteq A, B' \subseteq B$ of super-nodes of T_{low} with an edge of capacity c between A' and B' in T_{low}. Notice also that a pair s, t as required above can be chosen as long as $\mathcal{T}$ is not equal to T_{low}, and hence, together with the two inductive claims above, the Gomory-Hu execution continues until $\mathcal{T}$ becomes exactly T_{low}. □

We are now ready to prove our main theorem.

Proof of Theorem 4.1.2. Let $G = (V, E)$ be an input undirected graph with unit edge-capacities, and denote by V_{low} all the nodes in G whose degrees are at most the chosen parameter $d \in [n]$, and by $V_{high} = V \setminus V_{low}$ the nodes whose degrees are greater than d.

First use Lemma 4.2.2 to construct a d-partial tree T_{low}, and treat it as the auxiliary tree computed by a truncated execution of the Gomory-Hu algorithm. Then continue a Gomory-Hu execution (using this tree) to complete the construction of a cut-equivalent tree. Note that every node in V_{low} is in a singleton super-node of T_{low}, since its minimum cut value to any other node is at most d; thus a super-node V_i in T_{low} has more than one node if and only if it contains only nodes in V_{high}. Moreover, by the properties of T_{low}, two nodes have minimum-cut value greater than d if and only if they are in the same super-node V_i. Since by Lemma 4.2.3 there exists a truncated Gomory-Hu execution that produces T_{low}, a Gomory-Hu execution starting with T_{low} as the auxiliary tree will result in a cut-equivalent tree and the correctness follows. The running time bound follows as the first step of constructing T_{low} takes $\tilde{O}(md)$ time, and the second step of the Gomory-Hu execution takes $|V_{high}|$ invocations of $\mathsf{Max\text{-}Flow}$, that is running time $\sum_{i=1}^{m/d} T(m,n,F_i)$. Finally, we will use the following known claim regarding a bound on the total sum of capacities of cut-equivalent trees.

Claim 4.2.4 (From Lemma 4 in [BHKP07])**.** *Let G be a unit edge-capacity graph G and T its cut-equivalent tree. Then the sum of capacities over all edges of T is bounded by $2m$.*

Now, since every invocation of maximum st-flow with value F_i in our algorithm determines a unique edge with capacity F_i in the final cut-equivalent tree, and by Claim 4.2.4 the sum of the capacities over all the edges of the cut-equivalent tree satisfies $\sum_{i=1}^{m/d} F_i \leq 2m$, it holds that the total time spent on the m/d invocations of $\mathsf{Max\text{-}Flow}$ is bounded by $\Phi(m,n,d)$. Thus, the proof of Theorem 4.1.2 is concluded. $\square$

We use the $T(m,n,F) = \tilde{O}(m + m^{3/4}n^{1/4}F^{1/2})$ time algorithm by [ST18] to optimize our running time. By the concavity of $F^{1/2}$, the maximum of $\sum_{i=1}^{m/d} T(m,n,F_i)$ is attained when all $F_i = d$. By setting $d = \sqrt{m}n^{1/6}$ we get $\sum_{i=1}^{\sqrt{m}/n^{1/6}} (m + m^{3/4}n^{1/4}m^{1/4}n^{1/12}) = \sum_{i=1}^{\sqrt{m}/n^{1/6}} mn^{1/3} = m^{3/2}n^{1/6}$, which is faster than the known $\tilde{O}(mn)$-time algorithm of [BHKP07] whenever $m \in [n, n^{5/3}]$.

Finally, relying on a hypothetical $m^{1+o(1)}$-time algorithm for $\mathsf{Max\text{-}Flow}$, we could set $d = \sqrt{m}$ to get a total running time of $m^{1+o(1)} \cdot m/\sqrt{m} + \tilde{O}(m \cdot \sqrt{m}) \leq m^{3/2+o(1)}$, as claimed immediately after Theorem 4.1.2.

4.3 Near-Linear Nondeterministic Algorithm for Cut-Equivalent Tree

As no conditional lower bounds are known for the problem of constructing a cut-equivalent tree, one potentially promising approach is to design a reduction from SAT to prove that running time $n^{1+\delta-o(1)}$, for a fixed $\delta > 0$, is not possible assuming SETH. However, in this section we show that the existence of such a reduction (at least in the case of unit edge-capacities) would refute NSETH. This proves our Theorem 4.1.3.

Our main technical result in this section (Theorem 4.3.2) is a fast *nondeterministic* algorithm for constructing a cut-equivalent tree (the meaning of this notion will be formalized shortly). We then reach the conclusion about NSETH by following an argument first made in [CGI$^+$16], however we have to rewrite their argument (rather than use their definitions and results directly), in order to adapt it from decision problems or functions (where each input has exactly one output) to total search problems, since every graph has at least one cut-equivalent tree (see Section 4.3.2).

Generally speaking, a search problem P is a binary relation, and we say that S is a solution to instance x iff $(x,S) \in P$. Let $\mathrm{SOL}(x) = \{S : (x,S) \in P\}$ denote the set of solutions for instance x. We say that P is a *total search problem* [7] if every instance x has at least one solution, i.e., $\mathrm{SOL}(x) \neq \emptyset$. Let $\perp$ be the "don't know" symbol and assume that $\perp \notin \mathrm{SOL}(x)$ for all x. For example, in our problem of constructing a cut-equivalent tree, x is a graph and $\mathrm{SOL}(x)$ is the set of all cut-equivalent trees for x.

Definition 4.3.1 (Nondeterministic complexity of a total search problem)**.** *We say that a total search problem P has nondeterministic time complexity $T(n)$ if there is a deterministic Turing Machine M such that for every instance x of P with size $|x| = n$:*

1. $\forall g, \mathsf{T}_{\mathsf{HALT}}(M(x,g)) \leq T(n)$*, i.e., the time complexity of M on input x with guess g is bounded by $T(n)$;*

2. $\exists g, M(x,g) \in \mathrm{SOL}(x)$*, i.e., at least one guess leads M to output a solution;*

3. $\forall g, M(x,g) \in \{\perp\} \cup \mathrm{SOL}(X)$*, i.e., every guess leads M to output either a solution or "don't know".*

Note that the time bound in item 1 does not depend on g, which is useful for our purpose.

We can now state the main technical result of this section. We prove it in Section 4.3.1, and then use it in Section 4.3.2 to prove Theorem 4.1.3.

Theorem 4.3.2. *The nondeterministic complexity of constructing a cut-equivalent tree for an input graph with unit edge-capacities is $\tilde{O}(m)$, where m is the number edges in the graph.*

This algorithm employs the Gomory-Hu algorithm in a very specific manner, where the vertices chosen at each iteration are "centroids" (see below). The same choice was previously used by Anari and Vazirani [AV18] in the context of parallel algorithms (for planar edge-capacitated graphs), to achieve a logarithmic recursion depth, which is key for parallel time. However, since our goal is different (we want near-linear total time) we have to worry about additional issues, besides the depth of the recursion. Many auxiliary graphs must be handled throughout the execution of the algorithm, and for each one we need to verify multiple minimum cuts. This is done by guessing cuts and flows, and the main challenge is to argue that the total size of all these objects (the auxiliary graphs, and the cuts and flows within them) is only $\tilde{O}(m)$. Towards overcoming this challenge, we show a basic structural result about cut-equivalent trees (see Claim 4.3.9 below) which may have other applications. Prior to our work, it seemed unlikely that the Gomory-Hu approach could come close to near-linear time, even if Max-Flow could be computed in linear time, since a Max-Flow computation is executed many times in many auxiliary graphs. However, our analysis shows that the total size of all these auxiliary graphs can be near-linear (if the right vertices are chosen at each iteration), giving hope that this approach may still achieve the desired upper bound.

4.3.1 The Nondeterministic Algorithm

We now prove Theorem 4.3.2. Let $G = (V, E)$ be the input graph, and let $n = |V|$ and $m = |E|$.

[7] In some of the previous literature it is called a *total function*, although it is actually a relation rather than a function.

Overview. At a high level, the nondeterministic algorithm first guesses nondeterministically a cut-equivalent tree $\mathcal{T}^*$, and then verifies it by a (nondeterministic) process that resembles an execution of the Gomory-Hu algorithm that produces $\mathcal{T}^*$. Similarly to the actual Gomory-Hu algorithm, our verification process is iterative and maintains a tree $\mathcal{T}$ of super-nodes, which means, as described in Section 4.2, that every tree node i is associated with $V_i \subseteq V$, and these super-nodes form a partition $V = V_1 \sqcup \cdots \sqcup V_l$. This tree $\mathcal{T}$ is initialized to have a single super-node corresponding to V and then modified at each iteration, hence we shall call it the *intermediate tree*. If all guesses work well, then eventually every super-node is a singleton and the tree $\mathcal{T}$ corresponds to $\mathcal{T}^*$. Otherwise (some step in the verification fails), the algorithm outputs $\perp$.

In a true Gomory-Hu execution, every iteration partitions some super-node into exactly two super-nodes connected by an edge (say $V_i = S \sqcup T$). In contrast, every iteration of our verification process partitions some super-node into multiple super-nodes that form a star topology, whose center is a singleton (say $V_i = \{w\} \sqcup V_{i,1} \sqcup \cdots \sqcup V_{i,d}$, where super-node $\{w\}$ has edges to all super-nodes $V_{i,1}, \ldots, V_{i,d}$). We call this an *expansion step* (see Figure 4.2), and the node in the center of the star (i.e., w) the *expanded node*. These expansion steps will be determined from the guess $\mathcal{T}^*$. For example, in the extreme case that $\mathcal{T}^*$ itself is a star, our verification process will take only one expansion step instead of $|V| - 1$ Gomory-Hu steps.

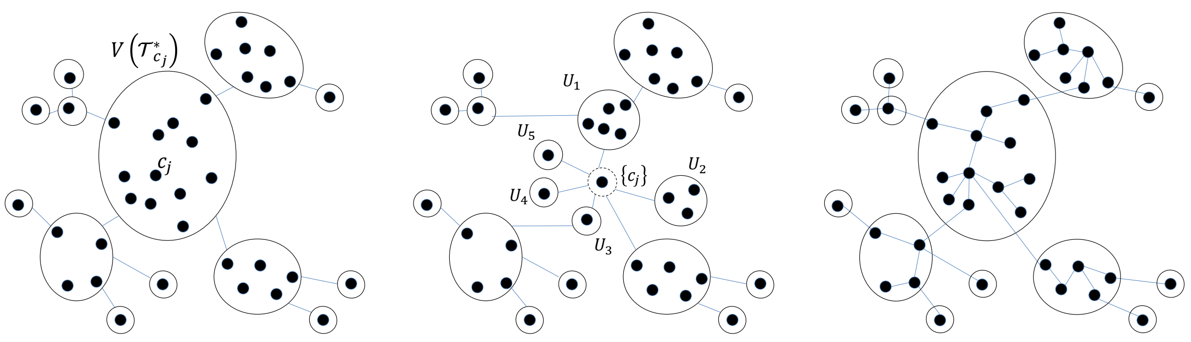

Figure 4.2: An illustration of the verification of a guessed tree $\mathcal{T}^*$. Left: the intermediate tree $\mathcal{T}$ right before an expansion step of the node c_j in the super-node $V(\mathcal{T}^*_{c_j})$. Middle: after the expansion step (of c_j, in the dashed circle) where $U_1, ..., U_4$ are c_j's neighbors in $\mathcal{T}^{(j+1)}$ such that $\bigcup_{i=1}^{4} U_i \cup \{c_j\} = V(\mathcal{T}^*_{c_j})$. Right: the guessed cut-equivalent tree $\mathcal{T}^*$.

To prove that our algorithm is correct, we will show that every expansion step corresponds to a valid sequence of steps in the Gomory-Hu algorithm. As the latter relies on minimum-cut computations in some auxiliary graph G', also our verification will need minimum-cut computations, which can be easily performed in nondeterministic linear time. However, this will not achieve overall running time $\tilde{O}(m)$, because in some scenarios (e.g., in the above example where $\mathcal{T}^*$ is a star), most of the $|V| - 1$ minimum-cut computations are performed on an auxiliary graph G' of size that is comparable to G, i.e., $\Omega(m)$. We overcome this obstacle using two ideas. First, we compute simultaneously all the minimum-cuts of the same expansion step in nondeterministic time that is *linear* in the size of G'. Second, we design a specific sequence of expansion steps such that the total size of *all auxiliary graphs* G' is $\tilde{O}(m)$.

Detailed Algorithm. The algorithm first guesses nondeterministically an edge-capacitated tree $\mathcal{T}^*$, and then verifies, as explained below, that it is a cut-equivalent tree. Here, verification means that upon the failure of any step, e.g., verifying some equality (say between the cut and flow values), the algorithm terminates with output $\perp$. (By the same reasoning, we may assume that all guesses are proper, e.g., a guessed tree is indeed a tree). The verification process starts by picking a sequence of nodes $c_0, c_1, c_2, \dots$ using the guess $\mathcal{T}^*$, as follows. Recall that a *centroid* of a tree is a node whose removal disconnects the tree into connected components (subtrees), each containing at most half the nodes in the tree. It is well-known that in every tree, a centroid exists and can be found in linear time. In a *recursive centroid decomposition* of a tree, one finds a centroid of the given tree, removes it and then repeats the process recursively in every connected component, until all remaining components are singletons (have size one). Our verification process computes this decomposition for the guess $\mathcal{T}^*$, which takes time $O(n \log n)$. For each recursion depth $i \geq 0$ (where clearly $i \leq \log n$), denote the set of centroids computed at depth i by $D_i \subset V$. For example, D_0 contains exactly one centroid, of the entire $\mathcal{T}^*$. Now let $c_0, c_1, c_2, \dots$ be the centroids in this decomposition in order of increasing depth, i.e., starting with the one centroid $c_0 \in D_0$, followed by the centroids from D_1 (ordered arbitrarily), and so forth. Let $\mathcal{T}^*_{c_j}$ be the subtree of $\mathcal{T}^*$ in which the centroid c_j was computed; for example $\mathcal{T}^*_{c_0} = \mathcal{T}^*$.

Observation 4.3.3. *For every two centroids from the same depth, namely, $c_j \neq c_{j'} \in D_i$, the corresponding subtrees $\mathcal{T}^*_{c_j}$ and $\mathcal{T}^*_{c_{j'}}$ are node disjoint.*

The verification process now initializes a tree $\mathcal{T}$, called the intermediate tree, to consist of a single super-node associated with V, and then performs on it expansion steps for nodes $c_0, c_1, c_2, \dots$ (in this order) as explained below.

We now explain how to perform an expansion step for node c_j. Recall that c_j is a centroid of the subtree $\mathcal{T}^*_{c_j}$, therefore it defines a partition $V(\mathcal{T}^*_{c_j}) = \{c_j\} \sqcup U_1 \sqcup \cdots \sqcup U_d$, where $U_1, \dots, U_d$ are the connected components after removing c_j. Notice that $d = \deg_{\mathcal{T}^*_{c_j}}(c_j) \leq \deg_{\mathcal{T}^*}(c_j)$, and that each U_k, $k \in [d]$, contains exactly one node $u_k \in U_k$ that is a neighbor of c_j in $\mathcal{T}^*_{c_j}$. The expansion step replaces the super-node $V(\mathcal{T}^*_{c_j})$ in $\mathcal{T}$ with $d+1$ super-nodes $\{c_j\}, U_1, \dots, U_d$. (We slightly abuse notation and use a subset of nodes like $V(\mathcal{T}^*_{c_j})$ also to refer to the super-node in $\mathcal{T}$ associated with this subset.) These $d+1$ new super-nodes are connected by a star topology, where the singleton $\{c_j\}$ at the center and each newly-added edge $(\{c_j\}, U_k)$ is set to the same capacity as the edge (c_j, u_k) in the guess $\mathcal{T}^*$. In addition, every edge that was incident to super-node $V(\mathcal{T}^*_{c_j})$, say $(V(\mathcal{T}^*_{c_j}), W)$, is modified to an edge (U, W), where U is one of the new super-nodes $\{c_j\}, U_1, \dots, U_d$, chosen according to the edge in $\mathcal{T}^*$ that was used to set a capacity for $(V(\mathcal{T}^*_{c_j}), W)$. (We will explain how the algorithm verifies the correctness of these edge weights shortly.)

It is easy to verify that the modifications to $\mathcal{T}$ (due to expansion steps) maintain the following property: Every super-node U in $\mathcal{T}$ induces a subtree of $\mathcal{T}^*$, i.e., the induced subgraph $\mathcal{T}^*[U]$ is connected. Moreover, eventually every super-node will be a singleton, and the intermediate tree will exactly match the guess $\mathcal{T}^*$. When we need disambiguation, we may use $\mathcal{T}^{(j)}$ to denote the tree's state before the expansion step for c_j. For example, $\mathcal{T}^{(0)}$ is the initial tree with a single super-node V.

Informally, the verification algorithm still has to check that the capacities of the newly-added tree edges correctly represent minimum-cut values. To this end, the algorithm now constructs an auxiliary graph G'_j just as in the Gomory-Hu algorithm (see Section 4.2).

Specifically, G'_j is constructed by taking G, and then for each connected component of $\mathcal{T}^{(j)} \setminus \{V(\mathcal{T}^*_{c_j})\}$ (i.e., after removing super-node $V(\mathcal{T}^*_{c_j})$ from $\mathcal{T}^{(j)}$), merging the nodes in (all the super-nodes in) this component into a single node. Our analysis shows (in Claim 4.3.6) that for all $s, t \in V(\mathcal{T}^*_{c_j})$, every minimum st-cut in the auxiliary graph G'_j is also a minimum st-cut in G. In addition, all the auxiliary graphs of a single depth q can be constructed in near-linear time (Lemma 4.3.10).

Observe that each neighbor u_k of c_j in $\mathcal{T}^*_{c_j}$ defines a (c_j, u_k)-cut in the auxiliary graph G'_j, given by the two connected components of $\mathcal{T}^* \setminus \{(c_j, u_k)\}$. The algorithm evaluates for each u_k the capacity of this cut in G'_j, and verifies that it is equal to the capacity of the newly-added edge $(\{c_j\}, U_k)$ (set to be the same as of edge (c_j, u_k) in $\mathcal{T}^*$). In fact, all these cuts evaluations are performed not sequentially but rather simultaneously for all $k \in [d]$, as follows. The key observation is that if we denote each aforementioned (c_j, u_k)-cut by $(V(G'_j) \setminus C'_k, C'_k)$, where $u_k \in C'_k$, then $\{c_j\}, C'_1, \ldots, C'_d$ are disjoint subsets of $V(G'_j)$. One can clearly evaluate the capacity of all these d cuts in a single pass over the edges of G'_j, and since each edge contributes to at most two cuts (by the disjointness), this entire pass takes only linear time $O(|E(G'_j)|)$.

Next, to verify that each (c_j, u_k)-cut exhibited above, namely, each $(V(G'_j) \setminus C'_k, C'_k)$, is actually a minimum (c_j, u_k)-cut in G'_j, the algorithm finds a flow whose value is equal to the cut capacity. In order to perform this task simultaneously for all $k \in [d]$, our verification algorithm employs a known result about disjoint trees, as a witness for maximum-flow values in a graph with unit edge-capacities (strictly speaking, this witness provides lower bounds on maximum-flow values). In the following theorem, a *directed tree rooted at* r is a directed graph arising from an undirected tree all of whose edges are then directed away from r. This is equivalent to an arborescence (having exactly one path from r to every node other than r), however we will not require that it spans all the graph nodes. In the following, $\mathsf{Max\text{-}Flow}_G(s, t)$ is the maximum st-flow value in a graph G.

Lemma 4.3.4. *Given an undirected multigraph $H = (V_H, E_H)$, a root node $r \in V_H$, and a function $\lambda : V_H \to [|E_H|]$, it is possible to nondeterministically verify in time $\tilde{O}(|E_H|)$ that*

$$\forall v \in V_H \setminus \{r\}, \qquad \mathsf{Max\text{-}Flow}_H(r, v) \geq \lambda(v). \tag{4.1}$$

Here, nondeterministic verification means that if (4.1) *holds then there exists a guess that leads to output "yes"; and if* (4.1) *does not hold then every guess leads to output "no".*

Proof. We use the following theorem known from [BFJ95, Theorem 2.7], in its variation from [CH03] as the Tree Packing Theorem.

Theorem 4.3.5. *Let H_e be an Eulerian directed graph, and r_e be a node in H_e. Then there exist $\max_{v \neq r_e}\{\mathsf{Max\text{-}Flow}_{H_e}(r_e, v)\}$ edge-disjoint directed trees rooted at r_e, such that each node $v \in H_e$ appears in exactly $\mathsf{Max\text{-}Flow}_{H_e}(r_e, v)$ trees.*

Given the undirected multigraph H, first subdivide each edge into two edges with a new node in between them, then orient each edge in both directions [8] to obtain an Eulerian directed graph H_e. Observe that the minimum-cut values between pairs of original nodes in H_e are the same as in H. Now find all maximum-flow lower-bound values from r in H_e by guessing $|V_H|$ edge-disjoint trees and then counting occurrences of each node in those trees.

[8]The subdivision and orientation are used to transform the undirected multigraph to a directed graph.

By Theorem 4.3.5, these counts correspond to maximum-flow lower-bound values from r. And so if the guessed trees support the values given by λ, then answer "yes", and otherwise answer "no". Note that the conversion to directed Eulerian graph multiplied the amount of edges by 2, and so the running time is still near linear. $\square$

The verification algorithm then applies Lemma 4.3.4 to G'_j with c_j as the root, and verifies in time $\tilde{O}(|E(G'_j)|)$ that the maximum-flow from c_j to each u_k is at least the capacity of the (c_j, u_k)-cut exhibited above (in turn verified to be equal to the capacity of edge (c_j, u_k) in $\mathcal{T}^*$).

Correctness. We begin by claiming that if the guessed tree $\mathcal{T}^*$ is a correct cut-equivalent tree of G, then our algorithm outputs $\mathcal{T}^*$; we discuss the complement case afterwards. Since $\mathcal{T}^*$ is a cut-equivalent tree, every verification step of an expansion will not fail and so the algorithm will not terminate prematurely, and output $\mathcal{T}^*$, as required.

Next, we show that if $\mathcal{T}^*$ is not a cut equivalent tree, then our algorithm will not succeed. This is proved mainly by the claim below, that an intermediate tree attained by expansion steps can be attained also by a sequence of Gomory-Hu steps.

Claim 4.3.6. *Suppose there is a sequence of Gomory-Hu steps producing tree $\mathcal{T}^{(j)}$, and that an expansion step is performed to produce $\mathcal{T}^{(j+1)}$. Then there is a sequence of Gomory-Hu steps that simulates also this expansion step and produces $\mathcal{T}^{(j+1)}$.*

Proof. Assume there is a truncated execution of the Gomory-Hu algorithm that produces $\mathcal{T}^{(j)}$. We describe a sequence of Gomory-Hu algorithm's steps starting with $\mathcal{T}^{(j)}$ that produces $\mathcal{T}^{(j+1)}$. Recall that $U_1, ..., U_d$ are $\{c_j\}$'s neighbors in $\mathcal{T}^{(j+1)}$ such that $\bigcup_{i=1}^{d} U_i \cup \{c_j\} = V(\mathcal{T}^*_{c_j})$, and $u_1, ..., u_d$ are the nodes by which the capacities of the edges $(\{c_j\}, U_k)$, $k \in [d]$, were chosen.

The Gomory-Hu steps are as follows, where we denote by $\mathcal{T}$ the intermediate tree along the execution. Starting with $\mathcal{T} = \mathcal{T}^{(j)}$, for $k = 1, ..., d$, the Gomory-Hu execution picks the pair c_j, u_k from the super-node containing it in $\mathcal{T}$ as the pair s, t in the Gomory-Hu algorithm description (see the description in Section 4.2), and the given minimum-cut value between them is asserted. Then, for the partitioning of this super-node in $\mathcal{T}$, the execution picks the minimum-cut between c_j, u_k as in $\mathcal{T}^*$ (which is a minimum cut also in the corresponding auxiliary graph) and modifies the intermediate tree accordingly. Note that the last expansion step was assumed to be successful (i.e., verified correctly), thus all the cuts chosen for the partitioning are minimum-cuts. $\square$

Now, assume for the contrary that $\mathcal{T}^*$ is not a cut-equivalent tree of G and our algorithm still produces it. As a consequence of Claim 4.3.6, there is a sequence of Gomory-Hu steps attaining $\mathcal{T}^*$, contradicting the proof of correctness of the Gomory-Hu algorithm (which cannot produce $\mathcal{T}^*$). Thus, it is impossible that our algorithm finishes and produces $\mathcal{T}^*$, and so in one of the minimum-cut verifications after an expansion step, the cut witness inspired from $\mathcal{T}^*$ would not be correct, or there would not be a set of directed trees to testify that the corresponding cuts are minimal. This completes the proof of correctness.

Running Time. Observe that the running time of a single expansion step, i.e., verifying its corresponding minimum cuts by evaluating cuts and flows, is near-linear in the size of the auxiliary graph. Thus, we only have to show that the total size of all the auxiliary graphs (over all the expansions) is near-linear in m. We prove in Lemma 4.3.7 below an $O(m)$ bound for a single depth q, and since the depth of the decomposition is $O(\log n)$, we immediately conclude in Corollary 4.3.8 that the total size of all auxiliary graphs over all depths is $\tilde{O}(m)$.

Lemma 4.3.7. *Let $D_q = \{c_{j_1}, \ldots, c_{j_2}\}$ contain the centroids at depth q. Then the total size of $G'_{j_1}, \ldots, G'_{j_2}$ is at most $O(m)$.*

Corollary 4.3.8. *The total size of all auxiliary graphs (over all depths) is $\tilde{O}(m)$.*

Proof of Lemma 4.3.7. Let us count for each edge $uv \in E(G)$ in how many auxiliary graphs of depth q it appears. This quantity turns out to be at most $2+(\text{dist}_{\mathcal{T}}(u,v)-1)$, where $\text{dist}_{\mathcal{T}}(u,v)$ is the hop-distance, i.e., the minimum number of edges (ignoring weights or capacities) in a path between u and v in the tree $\mathcal{T}$. The summand 2 comes from edges uv such that either u or v belong to $V(\mathcal{T}_{c_j})$ for some auxiliary graph G'_j. Clearly, every such edge is in at most two auxiliary graphs at depth q, because there is at most one index $j' \in D_q$ where $u \in V(\mathcal{T}_{c_{j'}})$ and at most one index $j'' \in D_q$ where $v \in V(\mathcal{T}_{c_{j''}})$. The summand $\text{dist}_{\mathcal{T}}(u,v) - 1$ bounds the other appearances of edge uv, i.e., when neither u nor v belongs to some $V(\mathcal{T}_{c_j})$, and is proved in the claim below. While our graph has unit capacities, the claim holds for general capacities.

Claim 4.3.9. *For every cut-equivalent tree $\mathcal{T}$ of a graph G with edge capacities $c_G : E \to \mathbb{R}_+$,*

$$\sum_{uv \in E(G)} c_G(u,v) \cdot \text{dist}_{\mathcal{T}}(u,v) \leq 2 \sum_{uv \in E(G)} c_G(u,v).$$

Proof. We first show that $\sum_{uv \in E_G} c_G(u,v) \cdot \text{dist}_{\mathcal{T}}(u,v) = \sum_{e \in E_{\mathcal{T}}} c_{\mathcal{T}}(e)$, where $c_{\mathcal{T}}(\cdot)$ denotes edge capacity in $\mathcal{T}$. Recalling that $\mathcal{T}$ is a cut-equivalent tree, each $c_{\mathcal{T}}(e)$ is the value of a certain cut in G, hence we can evaluate the right-hand side differently, by summing over the graph edges $uv \in E(G)$ and counting for each edge in how many such cuts it appears. Moreover, the count for each graph edge $uv \in E(G)$ is exactly $\text{dist}_{\mathcal{T}}(u,v)$ contributions of $c_G(u,v)$, giving altogether the left-hand side.

Second, we show that $\sum_{uv \in E(\mathcal{T})} c_{\mathcal{T}}(u,v) \leq 2 \sum_{uv \in E(G)} c_G(u,v)$ (see also Lemma 4 in [BHKP07]). To see this, observe that $c_{\mathcal{T}}(u,v) \leq \min\{\deg_{c_G}(u), \deg_{c_G}(v)\}$ where $\deg_{c_G}(u)$ is the total capacity of edges incident to u. Now fix a root vertex in $\mathcal{T}$, and bound each tree edge by $c_{\mathcal{T}}(u,v) \leq \deg_{c_G}(v)$, where v is the child of u (i.e., farther from the root) in $\mathcal{T}$. Summing this bound over all the tree edges and observing that the corresponding vertices v are all distinct proves the desired inequality, and the claim follows. □

To complete the proof of Lemma 4.3.7, recall that by Observation 4.3.3 the super-nodes $V(\mathcal{T}_{c_{j_1}}), \ldots, V(\mathcal{T}_{c_{j_2}})$ of the same depth q are pairwise disjoint. Thus, an edge uv appears in at most $\mathit{dist}_{\mathcal{T}^*}(u,v) - 1$ auxiliary graphs of depth q, which totals to $O(m)$ for all the edges in this depth according to the unit edge-capacity special case of the above Claim 4.3.9. This concludes Lemma 4.3.7. □

Next, we bound the time it takes to construct all the auxiliary graphs.

Lemma 4.3.10. *The total time it takes to construct the auxiliary graphs for all the expansions in the centroid decomposition is $\tilde{O}(m)$.*

Proof. Let c_j be a node that is expanded at some depth $q \geq 1$, and let $c_{j,1}, \dots, c_{j,d}$ be the expanded nodes in $U_1, \dots, U_d$, respectively at depth $q+1$ (or just $\perp$ for singletons).

Note that $G'_{j,1}, \dots, G'_{j,d}$ (whichever exist) can all be constructed in total time that is linear in the size of G'_j.

Thus, the total time it takes to construct the auxiliary graphs at a single depth q is linear in the size of the auxiliary graphs in the parent depth. Since the auxiliary graph at depth 0 (i.e., the entire graph) can trivially be constructed in $O(m)$ time and is linear in the input size, it follows by Corollary 4.3.8 that the total construction time of the auxiliary graphs for all depths takes at most $\tilde{O}(m)$ time. $\square$

4.3.2 Reduction from a Decision Problem to a total search problem

Let us start with the formal statement of NSETH.

Hypothesis 4.3.11 (Nondeterministic Strong Exponential-Time Hypothesis (NSETH))**.** *For every $\varepsilon > 0$ there exists $k = k(\varepsilon)$ such that k-TAUT (the language of all k-DNF formulas that are tautologies) is not in* $\mathrm{NTIME}(2^{n(1-\varepsilon)})$.

Note that deciding if a k-DNF formula is a tautology is equivalent to deciding if a k-CNF formula is satisfiable, thus the above hypothesis could be stated also using k-CNF appropriately. Next, we define (deterministic) fine-grained reductions from a decision problem to a total search problem. Note that these are Turing reductions.

Definition 4.3.12 (Fine-Grained Reduction from a Decision Problem to a total search problem)**.** *Let L be a language and P be a total search problem, and let $T_L(\cdot)$ and $T_P(\cdot)$ be time bounds. We say that (L, T_L) admits a fine-grained reduction to (P, T_P) if for all $\varepsilon > 0$ there is a $\gamma > 0$ and a deterministic Turing machine M^P (with an access to an oracle that generates a solution to every instance of P) such that:*

1. *M^P decides L correctly on all inputs when given a correct oracle for P.*
2. *Let $\tilde{Q}(M^P, x)$ denote the set of oracle queries made by M^P on input x of length n. Then the query lengths obey the bound*

$$\forall x, \qquad \mathsf{T}_{\mathsf{HALT}}(M^P, |x|) + \sum_{q \in \tilde{Q}(M,x)} (T_P(|q|))^{1-\varepsilon} \leq (T_L(n))^{1-\gamma},$$

where $\mathsf{T}_{\mathsf{HALT}}(M^P, |x|)$ is the maximal time M^P runs on inputs of size $|x|$.

We are now ready to prove the non-reducibility result under NSETH for total search problems with small nondeterministic complexity. The proof arguments are similar to those of Carmosino et al. [CGI+16].

Theorem 4.3.13. *Suppose P is a total search problem with nondeterministic time complexity $T(m)$. If for some $\delta > 0$ there is a deterministic fine-grained reduction from k-SAT with time-bound 2^n to P with time bound $T(m)^{1+\delta}$, i.e., from $(k\text{-SAT}, 2^n)$ to $(P, T(m)^{1+\delta})$, then NSETH is false.*

Proof. We will use the assumption of the theorem to describe a nondeterministic algorithm for k-TAUT that refutes NSETH. Let ϕ be an instance of k-TAUT, and note that $\phi \in k$-TAUT iff $\neg\phi \notin k$-SAT. Our nondeterministic algorithm A first computes the CNF formula $\neg\phi$, then simulates the assumed reduction M_1 from k-SAT to P on $\neg\phi$, and eventually outputs the negation of the simulation's answer, or *Reject* if the simulation returns $\perp$.

Let M_2 be the Turing Machine showing that P has nondeterministic time complexity $T(m)$. Whenever the reduction M_1 produces a query to P, our algorithm A executes M_2 on this query with some guess string g. Let g_i be the guess string used for the i^{th} query to P made by M_1. If any of the executions of M_2 throughout the simulation outputs $\perp$, then A stops and outputs *Reject*. Otherwise (all executions output valid answers), the simulation continues until M_1 terminates. At this point, the output of M_1 must be correct, and our algorithm A outputs the opposite answer.

Let us argue about the correctness of our algorithm. First, it only outputs *Accept* if the guesses and all answers to the P-queries were correct and then M_1 rejected, meaning that $\neg\phi \notin k$-SAT i.e., $\phi \in k$-TAUT. Second, for every yes-instance $\phi \in k$-TAUT there is at least one sequence of guesses $g_1, g_2, \dots$ that makes A output *Accept*, due to the correctness of the reduction M_1 and the fact that M_2 nondeterministically computes P correctly. Finally, the running time of A can be upper bounded by

$$\mathsf{T}_{\mathsf{HALT}}(M_1) + \sum_{q \in \tilde{Q}(M_1,x)} T(|q|) \leq \mathsf{T}_{\mathsf{HALT}}(M_1) + \sum_{q \in \tilde{Q}(M_1,x)} T(|q|)^{(1+\delta)(1-\varepsilon)} \leq (2^n)^{1-\gamma}$$

for $0 < \varepsilon < \delta$ where the last inequality is due to the reduction from k-SAT to P, $\mathsf{T}_{\mathsf{HALT}}(M_1)$ is the time of operations done by M_1, $\tilde{Q}(M_1, x)$ is the queries made by M_1 to the P-oracle on an input x, and the last inequality follows for some $\gamma(\varepsilon) > 0$ because M_1 is a correct fine-grained reduction. Thus, A refutes NSETH. □

Since the construction of a cut-equivalent tree is a total search problem, and by Theorem 4.1.2 its nondeterministic complexity is $\tilde{O}(m)$, applying Theorem 4.3.13 implies that any deterministic reduction from SETH to the construction of a cut-equivalent tree that implies a lower bound of $\Omega(m^{1+\delta})$, for some $\delta > 0$, would refute NSETH, concluding Theorem 4.1.3.

4.4 Conditional Lower Bound for All-Pairs Max-Flow

In this section we prove a conditional lower bound for All-Pairs Max-Flow in undirected graphs with node capacities. Our construction is inspired by the one in [KT18b], which was designed for directed graphs with edge capacities, but we adapts it using our new trick described in the introduction. In fact, readers familiar with the reduction in [KT18b] may notice that we had to tweak it a little, making it simpler in certain ways but more complicated in others. This was necessary in order to apply our new trick successfully to it.

The starting point for our reduction is the 3OV problem.

Definition 4.4.1 (3OV). *Given three sets $U_1, U_2, U_3 \subseteq \{0,1\}^d$ containing n binary vectors each, over dimension d, decide if there is a triple (α, β, γ) of vectors in $U_1 \times U_2 \times U_3$, whose dot product is 0. That is, a triple for which for all $i \in [d]$ at least one of $\alpha[i], \beta[i], \gamma[i]$ is equal to 0.*

An adaptation of the reduction by Williams [Wil05] shows that 3OV cannot be solved in $O(n^{3-\varepsilon})$ time for any $\varepsilon > 0$ and $d = \omega(\log n)$, unless SETH is false (see [ABW15]). For us, it suffices to assume the milder conjecture that 3OV cannot be solved in $O(n^{3-\varepsilon})$ time when $d = n^{\delta}$, for all $\varepsilon, \delta > 0$. Refuting this conjecture has important implications beyond refuting SETH [GIKW17, ABDN18], e.g. it refutes the Weighted Clique Conjecture.

The high level structure of the reduction is the following: create three layers of nodes that correspond to the three sets of vectors, with additional two layers in between them that correspond to the coordinates. These additional layers help keep the number of edges small by avoiding direct edges between pairs of vectors. Among other things, we utilize the trick described in the introduction and set the capacity of the nodes in the leftmost and rightmost sides to be 1, while making the other capacities much larger. This way a flow would not gain too much from crisscrossing through these nodes. Formally, we prove the following.

Lemma 4.4.2. *3OV over vector sets of size n and dimension d can be reduced to All-Pairs Max-Flow in undirected graphs with $\Theta(n \cdot d)$ nodes, $\Theta(n \cdot d)$ edges, and node capacities in $[2n^2d]$.*

Since, as explained earlier, the 3OV conjecture is a consequence of SETH, then Theorem 4.1.1 immediately follows from Lemma 4.4.2, which is proved below.

Proof of Lemma 4.4.2. Given a 3OV instance F we construct a graph G with maximum flow size between some pair (among a certain set of pairs) bounded by a certain amount if and only if F is a yes instance. For simplicity, we first provide a construction that has some of the edges directed (only where we will specifically mention that), and then we show how to avoid these directions. In addition, some of the edges will be capacitated as well, however the amount of such edges is small enough so that subdividing them with appropriate capacitated nodes will work too without a significant change to the size of the constructed graph.

An Intermediate Construction with Few Directed Edges. To simplify the exposition, we start with a construction of a graph G' in which most of the edges are undirected, but some are still directed (see Figure 4.3).

Our final graph G will be very similar to G'. It will have the same nodes and edges except that all edges will be undirected and the capacities on the nodes will be a little different.

We construct the graph G' on N nodes $V_1 \cup V_2 \cup V_3 \cup A \cup B \cup \{v_B\}$. The layer V_1 contains a node α of capacity 1 for every vector $\alpha \in U_1$. V_2 contains $d+1$ nodes for every vector $\beta \in U_2$: d nodes of capacity 1 denoted by β_i for every $i \in [d]$, plus a node denoted by β' of capacity $d-1$. V_3 contains a node γ of capacity 1 for every vector γ in U_3. The intermediate layer A contains $2d$ nodes: two nodes C_i^0 and C_i^1 of capacity n for every coordinate $i \in [d]$. The other intermediate layer B contains a node C_i of capacity n for every coordinate $i \in [d]$. Finally, the auxiliary node v_B has capacity $n(d-1)$. With a slight abuse of notation, we will use the following symbols in the following ways: α will be either a node in V_1 or a vector in U_1; β will be a vector in U_2; γ will be either a node in V_3 or a vector in U_3; and C_i will be either a node in B or a coordinate in $[d]$. The usage will be clear from context.

The edges of the network will be defined as follows. First, we describe the edges that depend on the given 3OV instance.

- For every α and $i \in [d]$, we add a *directed* edge from α to C_i^0 if $\alpha[i] = 0$, and a directed edge from α to C_i^1 if $\alpha[i] = 1$.

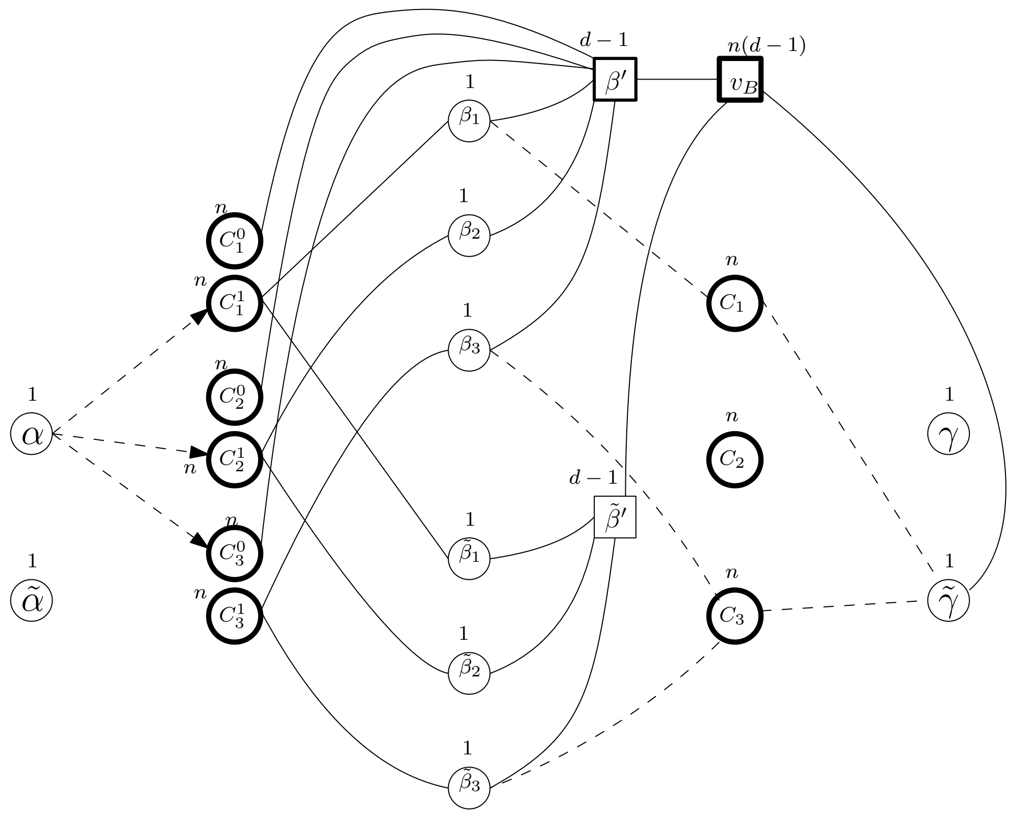

Figure 4.3: An illustration of part of the reduction. Here, U_1, U_2, and U_3 have two vectors each; α and $\tilde{\alpha}$ in U_1, β and $\tilde{\beta}$ in U_2, γ and $\tilde{\gamma}$ in U_3. Bolder nodes correspond to nodes of higher capacity, and dashed edges are conditional on the input instance. For simplicity, we omit the edges not relevant to α and $\tilde{\gamma}$, and also the edges from nodes in $\{C_i^0\}_{i\in[3]}$ to nodes in $\{\beta', \tilde{\beta}'\}$. In this illustration, $\alpha = 110$, $\beta = 101$, $\tilde{\beta} = 001$, and $\tilde{\gamma} = 101$. Note that the triple α, $\tilde{\beta}$, and $\tilde{\gamma}$ has an inner product 0, and indeed the maximum flow from α to $\tilde{\gamma}$ is $2 \cdot 3 - 1 = 5$.

- For every β, we add an (undirected) edge from β_i to C_i if $\beta[i] = 1$.
- For every γ and $i \in [d]$, we add an (undirected) edge from C_i to γ if $\gamma[i] = 1$.

Moreover, there will be some (undirected) edges that are independent of the vectors. For every β, we have an edge of capacity 1 from C_i^0 to β', and an edge of capacity 1 from C_i^1 to β_i. Also, for every β, we have an edge from β_i to β', and an edge from β' to v_B. Finally, for every γ, we have an edge from v_B to $\gamma \in V_3$. (Unless specified otherwise, these edges have no capacity constraints.)

The graph built has $N = n + 2d + n \cdot d + n + 1 + d + n = \Theta(nd)$ nodes, at most $O(nd)$ edges, all of its capacities are in $[N]$, and its construction time is $O(nd)$.

The following two claims prove the correctness of this intermediate reduction.

Claim 4.4.3. *If every triple of vectors in (U_1, U_2, U_3) has inner product at least 1, then for all pairs $\alpha \in V_1, \gamma \in V_3$ the maximum-flow in G' is at least $n \cdot d$.*

Proof. Assume that every triple of vectors in (U_1, U_2, U_3) has inner product at least 1, and fix some α and γ. We will explain how to send $n \cdot d$ units of flow from α to γ in G'. By the assumption, for every β there exist an $i \in [d]$ such that $\alpha[i] = \beta[i] = \gamma[i] = 1$, and denote this index by i_β. Each i_β induces a path $(\alpha \to C^1_{i_\beta} \to \beta_{i_\beta} \to C_{i_\beta} \to \gamma)$ from α to γ, and so we pass a single unit of flow through every one of them, in what we call the first phase. Note that so far, the flow sums up to n, and we carry on with describing the second phase of flow through nodes of the form β'.

We claim that for every β, an additional amount of $(d-1)$ units can pass through β', which would add up to a total flow of $n(d-1) + n = nd$, concluding the proof. Indeed, for every β, we send flow in the following way. For every $i \in [d] \setminus i_\beta$, if $\alpha[i] = 1$ then we send a single unit through $(\alpha \to C^1_i \to \beta_i \to \beta' \to v_B \to \gamma)$, and otherwise we send a unit of flow through $(\alpha \to C^0_i \to \beta' \to v_B \to \gamma)$.

Since we defined the flow in paths, we only need to show that the capacity constraints are satisfied. Nodes of the form C_i are only used in the first phase, and the flow through them equals n in total, and so their flow is within the capacity. The node v_B is only used in the second phase and has $n(d-1)$ units of flow passing through it, just as its capacity. For every β and $i = i_\beta$, we pass in the first phase a single unit of flow through β_i. For every β and $i \neq i_\beta$, we transfer in the second phase a unit of flow through β_i if and only if $\alpha[i] = 1$, thus it is bounded. For every β', we pass in the second phase exactly $(d-1)$ units of flow through β', preserving its capacity. For every $C^j_i \in N(\alpha)$, where for a node x we denote by $N(x)$ the set of nodes adjacent to x, with $i \in [d]$ and $j \in \{0, 1\}$, we pass a total of n units of flow to nodes in V_2, one unit on each edge, thus the capacities are preserved, concluding the proof. □

Claim 4.4.4. *If there is a triple of vectors $(\alpha_\Phi, \beta_\Phi, \gamma_\Phi) \in (U_1, U_2, U_3)$ whose inner product is 0, then the maximum-flow in G' from $\alpha_\Phi \in V_1$ to $\gamma_\Phi \in V_3$ is at most $nd - 1$.*

Proof. Assume for contradiction that there exists such a flow of value at least nd, and denote it by f. Let $f = \{p_1, ..., p_{|f|}\}$ be a description of f as a (multi-)set of paths of single units of flow. By our construction, the total capacity of all nodes in $N(\alpha_\Phi)$ sums up to nd exactly. Therefore, f must have all of the nodes in $N(\alpha_\Phi)$ saturated.

Consider a node $C^j_i \in N(\alpha_\Phi)$ for some $i \in [d]$ and $j \in \{0, 1\}$. Note that C^j_i is saturated in f while its capacity is n and it has exactly n edges adjacent to it (excluding the edges incoming from V_1) of capacity 1 each. Therefore, we get that every node in $N(C^j_i) \setminus V_1$ must receive a single unit of flow from C^j_i in f. Hence, every β-cloud, which we define as all the nodes that are associated with a β, must have exactly d flow paths in f for which it is the first β-cloud that they pass through. We call this a *first passing* of a path through a β-cloud. In particular, for every β and for every $i \in [d]$ such that $\alpha_\Phi[i] = 1$ there must be a path $p_{\beta,i}$ in f whose prefix is $(\alpha_\Phi, C^1_i, \beta_i, ...)$.

Our main claim is that the β_Φ-cloud can only have up to $d - 1$ flow paths that are first passing through it. Clearly, if there are more, then at least one of them does not pass through ${\beta_\Phi}'$ (whose capacity is only $d - 1$), so name this path p'. We will argue that this path must be in conflict with one of the $p_{\beta,i}$ paths described above.

For some $i \in [d]$ the prefix of p' must be $(\alpha_\Phi, C^1_i, \beta_{\Phi_i}, C_i, ...)$, since this is the only way it can avoid the node ${\beta_\Phi}'$. This can only happen for an $i \in [d]$ for which $\alpha[i] = \beta[i] = 1$, or else those edges will not exist in G. But since $(\alpha_\Phi, \beta_\Phi, \gamma_\Phi)$ is a triple whose inner product is 0, it must be that $\gamma_\Phi[i] = 0$ and so the edge $\{C_i, \gamma\}$ is not in the graph. Hence, after C_i this path can only go to a node $\tilde{\beta}_i$ for some $\tilde{\beta}$, and the (longer) prefix of p' must be $(\alpha_\Phi, C^1_i, \beta_i, C_i, \tilde{\beta}_i, ...)$.

Note that this is the same index i, and we know that $\alpha_\Phi[i] = 1$. Therefore, by the above, we know that there is another path $p_{\tilde{\beta},i}$ in f that has $\tilde{\beta}_i$ as the third node on the path. (That is, there is already a path that is first-passing through $\tilde{\beta}_i$.) This is a contradiction to the fact that the capacity of $\tilde{\beta}_i$ is 1. □

The Final Construction. The main issue with avoiding the directions on the edges between nodes in V_1 and A, is that additional α's might participate in the flow as well, potentially allowing one additional unit of flow to pass through. As described in the introduction, the solution is to multiply the capacities of all nodes that are not in $V_1 \cup V_3$ by $2n$. This is how we get our final graph G from G'. In the following we show how this modification concludes the proof of Lemma 4.4.2.

Claim 4.4.5. *If every triple of vectors in (U_1, U_2, U_3) has inner product at least 1, then for all pairs $\alpha \in V_1, \gamma \in V_3$ the maximum-flow in G is at least $2n^2d$.*

Proof. Since the flow that was defined in Claim 4.4.3 does not touch nodes in $V_1 \cup V_3$, considering the same flow in G but multiplied by $2n$, we get a new flow that is of size $nd \cdot (2n)$, concluding the proof. □

Claim 4.4.6. *If there is a triple of vectors $(\alpha_\Phi, \beta_\Phi, \gamma_\Phi) \in (U_1, U_2, U_3)$ whose inner product is 0, then the maximum-flow in G from $\alpha_\Phi \in V_1$ to $\gamma_\Phi \in V_3$ is at most $2n^2d - 1$.*

Proof. Let f be the maximum flow from α_Φ to γ_Φ in G. The paths in f can be divided into two kinds: paths that pass through nodes in $(V_1 \cup V_3) \setminus \{\alpha_\Phi, \gamma_\Phi\}$, and paths that do not. The total contribution of paths of the first kind can be upper bounded by the size of $(V_1 \cup V_3) \setminus \{\alpha_\Phi, \gamma_\Phi\}$, which is $2n - 2$, since the capacity of all nodes in this set is 1. On the other hand, paths from the second kind must obey the directions of the directed edges in G' and can therefore be used in G', except that in G their multiplicity (the amount of flow we push through them) can be larger by a factor of $2n$. Therefore, we can upper bound the total contribution of paths of the second kind by $2n$ times the maximum flow in G', which is $(nd - 1)(2n)$. Thus, the overall flow is at most $(nd - 1)(2n) + 2n - 2 = 2n^2d - 2$, which proves Claim 4.4.6. □

Since we showed a gap of at least one unit of flow between the yes and the no instances, the proof of Lemma 4.4.2 is concluded. □

4.5 Open Problems

Many gaps and open questions around the complexity of maximum flow remain after this work. We highlight a few for which our intuitions may have changed following our discoveries.

- Can we break the $O(mn)$ barrier also when the graphs have arbitrary (polynomial) capacities? Our result gives hope that this may be possible.
- Can we reduce the directed case to the undirected, node-capacitated case? Because of our lower bound, it is likely that both of these cases will end up having the same time complexity, and so such a reduction may be possible.

- Can we generalize the nondeterministic algorithm to arbitrary edge capacities? Notice that one obstacle for achieving that goal is finding lower bounds witnesses for flows from a certain source to other nodes.
- Can we prove any conditional lower bound for All-Pairs Max-Flow in undirected graphs with edge capacities? This is obviously the most important and intriguing open question in this context. Our new deterministic and nondeterministic upper bounds make this task more challenging than previously thought.

4.6 Acknowledgements

We would like to thank Arturs Backurs for asking about the nondeterministic complexity of the problems, Marvin Kunnemann for pointing out the connection to certifying algorithms, and Richard Peng for helpful comments on the different known upper bounds for Max-Flow.

Chapter 5

Cut-Equivalent Trees are Optimal for Min-Cut Queries[1]

5.1 Introduction

Minimum *st*-cut queries, or Min-Cut queries for short, are ubiquitous: Given a pair of nodes s, t in a graph G we ask for the minimum cut that separates them. Countless papers study their algorithmic complexity from various angles and in multiple contexts. Unless stated otherwise, we are in the standard setting of an undirected graph $G = (V, E, c)$ with $n = |V|$ nodes and $m = |E|$ weighted edges, where the weights (aka capacities) are polynomially bounded, i.e., $c : E \to \{1, \ldots, U\}$ for $U = \text{poly}(n)$. While a *Min-Cut query* asks for the set of edges of the minimum cut, a *Max-Flow query* only asks for its weight. [2] A single Min-Cut or Max-Flow query can be answered in time $\tilde{O}(m\sqrt{n})$ [LS14], [3] and there is optimism among the experts that near-linear time, meaning $\tilde{O}(m)$, can be achieved.

In the *data structure* (or *online*) setting, we would like to preprocess the graph once and then quickly answer queries. There are two naive strategies for this. We can either skip the preprocessing and use an offline algorithm for each query, making the query time at least $\Omega(m)$. Or we can precompute the answers to all possible $O(n^2)$ queries, making the query time $O(1)$, at the cost of increasing the time and space complexity to $\Omega(n^3)$ or worse.

Half a century ago, Gomory and Hu gave a remarkable solution [GH61]. By using an algorithm for a single Min-Cut query $n - 1$ times, they can compute a *cut-equivalent tree* (aka Gomory-Hu tree) of the original graph G. This is a tree on the same set of nodes as G, with the strong property that for every pair of nodes $s, t \in V$, their minimum cut in the tree is also their minimum cut in the graph. [4] This essentially reduces the problem from arbitrary graphs to trees, for which queries are much easier — the minimum *st*-cut is attained by cutting a single edge, the edge of minimum weight along the unique *st*-path, which can be reported in logarithmic time. [5] Cut-equivalent trees have other attractive properties beyond making queries faster, as they also provide a deep structural understanding of the graph by

[1] This chapter is based on [AKT20a].

[2] This terminology is common in the literature, although some recent papers [BSW15, BENW16] use other names.

[3] The notation $\tilde{O}(\cdot)$ hides $\text{poly}\log n$ factors (and also $\text{poly}\log U$ factors in our case of $U = \text{poly}(n)$).

[4] If G has a unique minimum *st*-cut then the reverse direction clearly holds as well.

[5] This immediately answers Max-Flow queries in logarithmic time. For Min-Cut queries extra work is required to output the edges in amortized logarithmic time; one simple way for doing it is shown in Section 5.4.

compressing all its minimum cut information into $O(n)$ machine words, and in particular they give a data structure which is space-optimal, as $\Omega(n)$ words are clearly necessary. Let us clarify that a cut-equivalent tree guarantees that for all $s, t \in V$, every edge e_{st} that has minimum weight along the tree's unique st-path, not only has the same weight as a minimum st-cut in G, but this edge also bipartitions the nodes into $V = S \sqcup T$ (the two connected components when e_{st} is removed from the tree), such that (S, T) is a minimum cut in the graph G. Without this additional property we would only have a weaker notion called a *flow-equivalent tree.*

Gomory and Hu's solution ticks all the boxes, except for the preprocessing time. Using current offline algorithms for each query [LS14], the total time for computing the tree is $\tilde{O}(mn^{3/2})$, and no matter how much the offline upper bound is improved, this strategy has a barrier of $\Omega(mn)$. While this barrier was not attained (let alone broken) for general inputs, there has been substantial progress on special cases of the problem. If the largest weight U is small, one can use offline algorithms [Mad16, LS20a, LS20b] that run in time $\tilde{O}(\min\{m^{10/7}U^{1/7}, m^{11/8}U^{1/4}, m^{4/3}U^{1/3}\})$ to get even closer to the barrier. In the unweighted case (i.e., unit-capacity $U = 1$), Bhalgat, Hariharan, Kavitha, and Panigrahi [BHKP07] (see also [KL15]) achieved the bound $\tilde{O}(mn)$ without relying on a fast offline algorithm, and this barrier was partially broken recently with a time bound of $\tilde{O}(m^{3/2}n^{1/6})$ [AKT20b]. Near-linear time algorithms were successfully designed for planar graphs [BSW15] and surface-embedded graphs [BENW16]. See also [GT01] for an experimental study, and the Encyclopedia of Algorithms [Pan16] for more background.

Meanwhile, on the hardness side, the only related lower bounds are for the online problem in the harder settings of directed graphs [AWY18, KT18b, AGI$^+$19] or undirected graphs with node weights [AKT20b], where Gomory-Hu trees cannot even exist, because the $\Omega(n^2)$ minimum cuts might all be different [HL07]. However, no nontrivial lower bound, i.e., of time $\Omega(m^{1+\varepsilon})$, is known for computing cut-equivalent trees, and there is even a barrier for proving such a lower bound under the popular Strong Exponential-Time Hypothesis (SETH) at least in the case of unweighted graphs, due to the existence of a near-linear time *nondeterministic* algorithm [AKT20b]. Thus, the following central question remains open.

Open Question 1. *Can one compute a cut-equivalent tree of a graph in near-linear time?*

A seemingly easier question is to design a data structure with near-linear time preprocessing that can answer queries in near-constant (which means $\tilde{O}(1)$, i.e., polylogarithmic) time. We should clarify that we are interested in near-constant *amortized* time; that is, if the output minimum st-cut has $k_{s,t}$ edges then it is reported in time $\tilde{O}(k_{s,t})$. Building cut-equivalent trees is one approach, but since they are so structured they might be limiting the space of algorithms severely.

Open Question 2. *Can one preprocess a graph in near-linear time to answer Min-Cut queries in near-constant amortized time?*

An even simpler question is the *single-source* version, where the data structure answers only queries $s, t \in V$ where s is a fixed source (i.e., known at preprocessing stage) and t can be any target node. This restriction seems substantial, as the number of possible queries goes down from $O(n^2)$ to $O(n)$, and in several contexts the known single-source algorithms are much faster than the all-pairs ones. One such context is shortest-path queries, where single-source is solved in near-linear time via Dijkstra's algorithm, while the all-pairs problem is conjectured to be cubic. Another context is Max-Flow queries in *directed* graphs(digraphs), where single-source is trivially solved by $n-1$ applications of Max-Flow, while based on some conjectures, all-pairs requires at least $\Omega(n^{3/2})$ such applications [KT18b, AGI+19]. Single-source Max-Flow queries is currently faster than all-pairs also in the special case of unit-capacity DAGs [CLL13]. However, this is still open for undirected Min-Cut queries.

Open Question 3. *Can one preprocess a graph in near-linear time to answer Min-Cut queries from a single source s to any target $t \in V$ in near-constant amortized time?*

It is natural to suspect that each of these questions is strictly easier than the preceding one. The case of bounded-treewidth graphs gives one point of evidence since a positive solution to Question 2 (and thus 3) was found over two decades ago [ACZ98], but Question 1 remained open to this day.

5.1.1 Our Results

Our first main contribution is to prove that all three open questions above are *equivalent.* We can extract a cut-equivalent tree from any data structure, even if it only answers single-source queries, without increasing the construction time by more than logarithmic factors. Thus, the appealingly simple trees are near-optimal as data structures for Min-Cut queries in all efficiency parameters; we find this conclusion quite remarkable.

Informal Theorem 1. *Cut-equivalent trees can be constructed in near-linear time* ***if and only if*** *there is a data structure with near-linear time preprocessing and $\tilde{O}(1)$ amortized time for Min-Cut queries,* ***and even if*** *the queries are restricted to a fixed source.*

The main new link that we establish in this chapter is to reduce Question 1 to Question 3, by essentially designing an entirely new algorithm for constructing cut-equivalent trees. The precise statement is given in Theorem 5.3.1. The two other links required for the equivalence are from Question 3 to Question 2, which holds by definition, and from Question 2 to Question 1. The latter link is to be expected, and was shown before in specific settings; for completeness, we give a simple proof via 2D range-reporting in Theorem 5.4.1. Thus, we get the reduction from all-pairs to single-source indirectly by going through the trees, and we are not aware of another way to prove this counter-intuitive link.

Notably, our result holds not only for general graphs but also for every graph family closed under minors. It is particularly useful for bounded-treewidth graphs, for which the two-decades-old results of Arikati, Chaudhuri, and Zaroliagis [ACZ98] now imply the construction of a cut-equivalent tree in near-linear time, as stated below. We do not see an alternative way to compute a cut-equivalent tree, e.g., using directly the techniques of [ACZ98], where parts of the graph G are replaced by constant-size mimicking networks [HKNR98].

Corollary 5.1.1 (see Corollary 5.3.2)**.** *A cut-equivalent tree for a bounded-treewidth graph G can be constructed in randomized time $\tilde{O}(m)$.*

In planar graphs, combining our reduction with the single-source algorithm of [LNSW12] gives an alternative to the all-pairs algorithm of [BSW15] that used a very different technique. [6]

[6]The conference paper of [BSW15] appeared in FOCS 2010, before [LNSW12] appeared in FOCS 2012. While the latter solves an easier task (single-source), it does so for the harder setting of *directed* planar graphs.

To evaluate our results, consider how much other existing techniques for constructing cut-equivalent trees would benefit from a (hypothetical) data structure for Min-Cut queries. The classical Gomory-Hu algorithm would have two main issues. First, it modifies the graph (merging some nodes) after each Min-Cut query, hence preprocessing a single graph (or a few ones) cannot answer all the $n-1$ queries. This issue was alleviated by Gusfield [Gus90], who modified the Gomory-Hu algorithm so that all the $n-1$ queries are made on the original graph G. A second issue is that the answer to each query might have $\Omega(m)$ edges, hence the total time $\Omega(mn)$ would far exceed $\tilde{O}(m)$. Optimistically, a more careful analysis could give an upper bound of $O(\phi)$, where ϕ is the total number of edges (in the original graph) in the $n-1$ cuts corresponding to the final tree's edges. Clearly, any such algorithm that does not merge edges must take $\Omega(\phi)$ time. Still, in weighted graphs ϕ could be $\Omega(mn)$, and even bounded-treewidth graphs could have $\phi = \Omega(n^2)$ even though $m = O(n)$ (e.g., a path with an extra node connected to all others). Therefore, our approach, which is very different from Gusfield's, shaves a factor of n. Notably, our result does not apply if the data structure is available only for unweighted graphs, because we need to perturb the edge weights to make all minimum cuts unique; but in this unweighted setting $\phi = O(m)$ [BHKP07, Lemma 5], hence it is plausible that other techniques, e.g. [Gus90, KL15], would be capable of showing the equivalence.

It is worth mentioning in this context a somewhat restricted form of the equivalence in unweighted graphs. In this case, the known $\tilde{O}(mn)$ time algorithm [BHKP07] for constructing a cut-equivalent tree actually runs in time $\tilde{O}(\phi \cdot c)$ where $c = \max_{u,v \in V} \mathsf{Max\text{-}Flow}(u,v)$ is at most n in unweighted graphs, utilizes a tree-packing approach [Gab95, Edm70] to find *minimal* Min-Cuts between a single source and multiple targets, meaning that the side not containing the source is minimal with respect to containment. Their method crucially relies on this minimality property to bypass the well-known barrier of uncrossing multiple cuts found in the same graph (which could be an auxiliary graph or the input G). This tree-packing approach is the basis of a few algorithms for cut-equivalent trees [CH03, HKP07, AKT20b], and it does not seem useful for weighted graphs.

While the equivalence for flows is incomparable to that for cuts, our techniques are robust enough to prove it. In particular, we show that $\tilde{O}(n)$ Max-Flow queries are sufficient to construct a flow-equivalent tree. Currently, this relaxation (flow-equivalent instead of cut-equivalent tree) is not known to make the problem easier in any setting, although Max-Flow queries could potentially be computed faster than Min-Cut queries. Our proof follows from a lemma that an n-point ultrametric can be reconstructed from $\tilde{O}(n)$ distance queries, under the assumption that it contains at least (and thus exactly) $n-1$ distinct distances (see Theorem 5.5.3). Interestingly, it is easy to show that without this extra assumption, $\Omega(n^2)$ queries are needed. To our knowledge, this is the first efficient construction of flow-equivalent trees only from Max-Flow queries (without looking at the cuts themselves). A well-known non-efficient construction (see [GH61]) is to make Max-Flow queries for all $O(n^2)$ pairs, view it as a complete graph with edge weights, and take a maximum-weight spanning tree.

Informal Theorem 2 (see Theorem 5.5.1)**.** *Flow-equivalent trees can be constructed in near-linear time **if and only if** there is a data structure with near-linear time preprocessing and $\tilde{O}(1)$ time for Max-Flow queries.*

(1+ε)-Approximations Our first result offers a quantitative improvement over the Gomory-Hu reduction from cut-equivalent trees to Min-Cut queries. It turns out that our technique also gives a qualitative improvement. A well-known open question among the experts, see e.g. [Pan16], is to utilize *approximate* Min-Cut queries (to construct an approximate cut-equivalent tree). An obvious candidate is an algorithm of Kelner et al. [KLOS14] for the offline setting (i.e., a single query), that achieves $(1+\varepsilon)$-approximation and runs in near-linear time. It beats the time-bound of all known exact algorithms, however no one has managed to utilize it for the online setting, or for constructing equivalent trees. It is not difficult to come up with counter-examples (see Section 5.2.1) that show that following the Gomory-Hu algorithm but using at each iteration a $(1+\varepsilon)$-approximate (instead of exact) minimum cut, results with a tree whose quality (approximation of the graph's cut values) is arbitrarily large. Our second main contribution is an efficient reduction from *approximate* equivalent trees to *approximate* Min-Cut queries. Previously, no such reductions were known (the aforementioned maximum-weight spanning tree would again give a non-efficient solution).

Informal Theorem 3 (see Theorem 5.2.1)**.** *Assume there is an oracle that can answer Min-Cut queries within $(1+\varepsilon)$-approximation. Then one can compute, using $\tilde{O}(n)$ queries to the oracle and an additional processing in time $\tilde{O}(n^2)$:*

1. *a $(1+\varepsilon)$-approximate flow-equivalent tree; and*
2. *a tree-like data structure that stores $\tilde{O}(n)$ cuts and can answer a Min-Cut query in time $\tilde{O}(1)$ and with approximation $1+\varepsilon$ by reporting (a pointer to) one of these stored cuts.*

For unweighted graphs, we can improve the $\tilde{O}(n^2)$ term to $\tilde{O}(m)$ which could be significant. While it may not be obvious why our new data structure is better than the oracle we start with, there are a few benefits (see Section 5.2.2). Most importantly, since it only uses $\tilde{O}(n)$ queries, we can combine our reduction with the algorithm of Kelner et al. [KLOS14] (even though it is for the offline problem, we essentially plug it into our reduction), and obtain three new approximate algorithms that are faster than state-of-the-art exact algorithms! We discuss these results next.

Corollary 5.1.2 (Section 5.2.2)**.** *Given a capacitated graph G on n nodes, one can construct a $(1+\varepsilon)$-approximate flow equivalent tree of G in randomized time $\varepsilon^{-4}\cdot n^{2+o(1)}$.*

It follows that the All-Pairs Max-Flow problem in undirected graphs can be solved within $(1+\varepsilon)$-approximation in time $n^{2+o(1)}$, which is optimal up to sub-polynomial factors since the output size is $\Omega(n^2)$. This problem is also well-studied in directed graphs [May62, Jel63, HL07, LNSW12, CLL13, GGI$^+$17], where it is known that exact solution in sub-cubic time is conditionally impossible [KT18b, AGI$^+$19], but it is open for approximated solutions.

Corollary 5.1.3 (Section 5.2.2)**.** *Given a capacitated graph G on n nodes, one can construct in $\varepsilon^{-4}\cdot n^{2+o(1)}$ randomized time, a data structure of size $\tilde{O}(n^2)$, that stores a set $\mathcal{C}$ of $\tilde{O}(n)$ cuts, and can answer a Min-Cut query in time $\tilde{O}(1)$ and with approximation $1+\varepsilon$ by reporting a cut from $\mathcal{C}$.*

Altogether, we provide for all three problems above (flow-equivalent tree, All-Pairs Max-Flow, and data structure for Max-Flow) randomized algorithms that run in time $n^{2+o(1)}$. Previously, the best approximation algorithm known for these three problems was to sparsify G into $m' = \tilde{O}(\varepsilon^{-2}n)$ edges in randomized time $\tilde{O}(m)$ using [BK15b] (or its generalizations), and then execute on the sparsifier the Gomory-Hu algorithm, which takes time $\tilde{O}(n\cdot m'\sqrt{n}) =$

$\tilde{O}(\varepsilon^{-2}n^{2.5})$. The best exact algorithms previously known for these problems was essentially to compute a cut-equivalent tree runs in time $O(mn^{1.5})$. An alternative way to approximate Max-Flow queries without the Gomory-Hu algorithm is to use Räcke's approach of a cut-sparsifier tree [Räc02]. This is a much stronger requirement (it approximates all cuts of G) and can only give polylogarithmic approximation factors. Its fastest version runs in near-linear time $m^{1+o(1)}$ and achieves approximation factor $O(\log^4 n)$ [RST14].

Unfortunately, we could not prove the same results for $(1+\varepsilon)$-*cut*-equivalent trees and more new ideas are required; in Section 5.2.1 we show an example where our approach fails. Interestingly, this is the first setting where we see different time bounds showing that the extra requirements of cuts indeed make the equivalent trees harder to construct.

Besides the inherent interest in the equivalence result and its applications, we believe that our results make progress towards the longstanding goal of designing optimal algorithms for cut-equivalent trees. It is likely that such algorithms will be achieved via a fast algorithm for online queries, as was the case for bounded-treewidth graphs.

5.1.2 Preliminaries

A *Min-Cut data structure* for a graph family $\mathcal{F}$ is a data structure that after preprocessing of a capacitated graph $G \in \mathcal{F}$ in time $t_p(m)$, can answer Min-Cut queries for any two nodes $s, t \in V$ in amortized query time (or output sensitive time) $t_{mc}(k_{st})$, where k_{st} denotes the output size (number of edges in this cut). This means that the actual query time is $O(k_{st} \cdot t_{mc}(k_{st}))$. A $(1+\varepsilon)$-approximate Min-Cut data structure is defined similarly but for $(1+\varepsilon)$-approximate minimum st-cut whose total capacity is at most $(1+\varepsilon)$ times that of the minimum st-cut in G. We denote by $\mathsf{Max\text{-}Flow}_G(s,t)$ the value of the minimum-cut between s and t, and we might omit the graph G subscript when it is clear from the context. Throughout, we restrict our attention to connected graphs and thus assume that $m \geq n-1$, and additionally we assume that the edge-capacities are integers (by scaling).

5.2 Our Approximation Algorithms

In this section we present our approximation algorithms, but first we give a high level overview of them.

5.2.1 Overview

Here we discuss the obstacles to speeding up Gomory-Hu's approach, and why plugging in *approximate* Min-Cut queries fails to produce an *approximate* cut-equivalent tree. To explain how our approach overcomes these issues, we present the key ingredients in our approximation algorithm from Section 5.2.2. This overview also prepares the reader for Section 5.3, which is the most complicated part of this chapter and proves our main result (Theorem 5.3.1).

Overview of the Gomory-Hu method Start with all nodes forming one super-node V. Then, pick an arbitrary pair of nodes s, t from the super-node, find a minimum st-cut $(S, V \setminus S)$, and split the super-node into two super-nodes S and $V \setminus S$. Then connect the two new super-nodes by an edge of weight $w(S, V \setminus S)$, and recurse on each of them. In each recursive call (which we also view as an iteration), say on a super-node V', the Min-Cut query is performed on an auxiliary graph $G_{V'}$ that is obtained from G by contracting every super-node other than V'. These contractions prevent the other super-nodes from being split by the cut, which is crucial for the consistency of the constructed tree, and by a key lemma about uncrossing cuts (proved using submodularity of cuts), these contractions (viewed as imposing restrictions on the feasible cuts in $G_{V'}$) do not increase the value of the minimum st-cut. The cut found in $G_{V'}$ is then used to split V' into two new super-nodes, and every edge that was incident to V' is "rewired" to exactly one of the new super-nodes. The process stops when every super-node contains a single node, which takes exactly $n-1$ iterations and results in a tree on n super-nodes, giving us a tree on V.

Why Gomory-Hu fails when using approximations There are two well-known issues (see [Pan16]) for employing this approach using *approximate* (rather than exact) Min-Cut queries, even if the approximation factor is as good as $1+\varepsilon$. The first issue is that errors of this sort multiply, and thus a $(1+\varepsilon)$-factor at each iteration accumulates in the final tree to $(1+\varepsilon)^d$, where d is the depth of the recursion. The second issue is even more dramatic; without the uncrossing-cuts property, the error could increase faster than multiplying and might be unbounded even after a single iteration. The reason is that when we find in super-node V' a cut $(S, V' \setminus S)$ that is (approximately) optimal for a pair $s,t \in V'$, we essentially assume that for all pairs $s' \in S, t' \in V \setminus S$ there is an (approximately) optimal cut that splits at most one of S and $V' \setminus S$ (not both). While true for exact optimality, it completely fails in the approximate case, and there are simple examples, see e.g. Figure 5.1, where allowing $(1+\varepsilon)$-approximation in the very first iteration makes the error of the final tree unboundedly large. We will refer to this issue as the main issue.

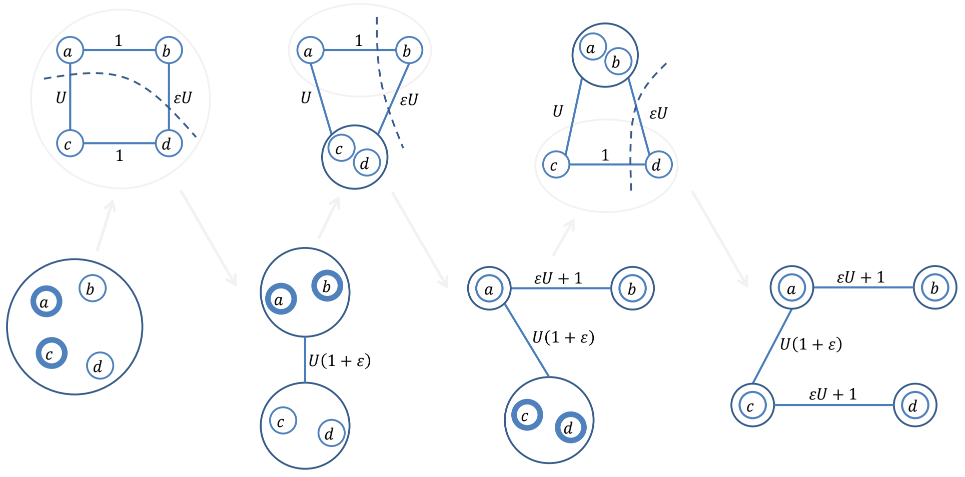

Figure 5.1: An example of the main issue with using $(1+\varepsilon)$-approximate minimum cuts in the Gomory-Hu algorithm. The input graph G is at the top left; the intermediate trees are at the bottom, from left to right; and the auxiliary graphs $G_{V'}$ are at the top. Each iteration uses a $(1+\varepsilon)$ Min-Cut for the node pair shown in bold. In the input graph $\mathsf{Max\text{-}Flow\text{-}Value}(b,c) = 2$ but in the tree it is $\Omega(U)$; thus the error can be as bad as $\text{poly}(n)$.

Our strategy Our approach is different and simultaneously resolves both issues for flow-equivalent trees; for cut-equivalent trees, as we show below, the first issue remains (but not the second).

Our main insight is to identify a property of the cut $(S, V' \setminus S)$, that is sufficient to resolve the main issue: This property is stronger than being a minimum *st*-cut, and requires that for all pairs $s' \in S, t' \in V' \setminus S$, this same cut is an (approximate) minimum $s't'$-cut, i.e., it works for them as well. Thus, the error for every pair s', t' from this split of V' is bounded by $(1+\varepsilon)$-factor, and we can recursively deal with pairs inside the same super-node. While this property may seem too strong, notice that it holds whenever $(S, V' \setminus S)$ is an (approximate) global minimum cut (i.e., achieves the minimum over all pairs $s', t' \in V'$). While our algorithm builds on this intuition, it does not compute a global minimum cut at each iteration, but rather employs a more complicated strategy that it is substantially more efficient. For example, its recursion depth is bounded by $O(\log n)$, which is important to bound the overall running time, and also to control the approximation factor.

Bounding the depth of the recursion The foremost idea is that the recursion depth should be bounded by $O(\log n)$. This does not happen in the Gomory-Hu algorithm, nor in the aforementioned strategy of using an (approximate) global minimum cut, where splits could be unbalanced and recursion depth might be $\Omega(n)$. Assuming – by way of wishful thinking – that the total time spent in all recursive calls of the same level is $\tilde{O}(m)$, [7] the challenge is to dictate how to (quickly) choose cuts so that the recursion depth is small.

Instead of insisting on a balanced cut, we partition the super-node V' into *multiple* sets at once, which can be viewed as performing a batch of consecutive Gomory-Hu iterations at the cost of one iteration (up to logarithmic factors). This approach was previously used in a few other algorithmic settings, however, none of their methods is applicable in our context. [8] Before explaining how our algorithm computes a partition, let us explain which properties it needs to satisfy. A partition of super-node V' into r sets $S_1, \ldots, S_r$ (that will be processed recursively) should satisfy the following strong property:

(*) For every pair $s' \in S_i, t' \in S_j$ for $i \neq j$, at least one of $(S_i, V' \setminus S_i)$ or $(S_j, V' \setminus S_j)$ corresponds in $G_{V'}$ to a $(1+\varepsilon)$-approximate minimum $s't'$-cut.

(We will actually allow an exception of one set S_0 that does not satisfy this property, and must be handled in a special way; this is the set V''_{big} in Section 5.2.3.) In addition, the sizes of these sets should be bounded by $|V'|/2$ (with the exception of the set S_0, which is bounded by $\frac{3}{4}|V'|$) which guarantees recursion depth $O(\log n)$, unlike a global minimum cut.

Our algorithm to partition V' picks a pivot node $p \in V'$ and queries a data structure built for $G_{V'}$ for an (approximate) minimum cut between p and every other node $u \in V'$; let $S_u \subset V'$ be the side of u in the returned cut. To form a partition out of these $|V'| - 1$ sets

[7] One moral justification is that super-nodes V' of the same recursion level are disjoint, as they form a partition of V. However, the real challenge is to process their auxiliary graphs $G_{V'}$. This may be possible in the special case where G is unweighted, becuase the total size (number of edges) of these auxiliary graphs (from one level) is $O(m)$ [BHKP07, BCH+08, KL15, AKT20b], but for a general graph G the total size of these auxiliary graphs might easily exceed $\tilde{O}(m)$.

[8] This approach was used in three different algorithmic settings: (1) in the special case of an unweighted graph G [BHKP07, BCH+08]; (2) in parallel algorithms [AV18], which can compute in parallel polynomially-many cuts (e.g., for all $s', t' \in V'$) to find a partition; or (3) in non-deterministic algorithms [AKT20b], which can "guess" a good partition but have to verify it quickly (achieved in [AKT20b] for an unweighted graph G).

S_u, reassign each node u to a set $S_{u'}$ that contains u, which naturally defines a partition (by grouping nodes reassigned to the same $S_{u'}$). The reassignment process is elaborate and subtle (see Section 5.2.3), aiming to preserve property (*) while reassigning nodes only to sets $S_{u'}$ of size at most $|V'|/2$.

Choosing effective pivots The above technique is not sufficient for bounding the depth of the recursion, because a poorly chosen pivot p might result in many unbalanced cuts (sets S_u of size larger than $\frac{3}{4}|V'|$), in which case this pivot is ineffective. Our next idea is that for a randomly chosen pivot $p \in V'$ this will not happen with high probability. [9] We analyze the performance of a random pivot using a simple lemma about tournaments that works as follows (see Lemma 5.2.4 and Corollary 5.2.5 for details). Assume for now that the Min-Cut data structure is deterministic (we show how to lift this assumption in Section 5.2.5), then every query $\{x, y\}$ (described as an unordered pair) is answered with some cut (S_x, S_y), and obviously $|S_x| \leq |V'|/2$ or $|S_y| \leq |V'|/2$ (or both). It follows by symmetry that a query for $\{u, p\}$ has a chance of at least $1/2$ of having $|S_u| \leq |V'|/2$, in which case we say that node u is "good" (in Section 5.2.2 we call these V_{small}). But we need a stronger property, that at least $1/4$ of the nodes in V' are good in this sense; we thus define on the nodes V' a tournament, with an edge directed from x to y whenever $|S_x| \leq |S_y|$, and prove that most nodes have a large out-degree, and will thus be effective pivots.

With constant probability, such an effective pivot is chosen, hence the number of nodes that are not good is bounded by $\frac{3}{4}|V'|$, and we must handle them with a separate recursive call (this is the problematic set V''_{big} in Section 5.2.3). A related but different issue that arises in Section 5.3.5 is that we cannot afford a Min-Cut query from p to all other $u \in V'$. To handle this we utilize the mentioned tournament properties by making Min-Cut queries from a random pivot p to only a small sample of targets.

Using dynamic-connectivity algorithms Even if the recursion depth is bounded by $O(\log n)$, it is not clear how to execute the entire algorithm in near-linear time, as each iteration computes $|V'| - 1$ cuts followed by a reassignment process. A straightforward implementation could require quadratic time $\Omega(n^2)$ even in the first iteration (on super-node V), which appears to be necessary because in some instances the total size of all good sets S_u (where $|S_u| \leq n/2$) is indeed $\Omega(n^2)$. For unweighted graphs, however, the total number of *edges* in these cuts (all minimum cuts from a fixed source to all targets) can be bounded by $O(m)$ (see Lemma 4 in [BHKP07], and Lemma 5.2.8 ahead), and indeed in this case our entire algorithm can be executed in time $\tilde{O}(m)$. The key is to only spend time proportional to the number of edges in each cut, rather than to the number of nodes $|S_u|$. In unweighted graphs, and also in the "capacitated auxiliary graphs" that we construct in Section 5.3, the total number of nodes and edges our algorithm observes is bounded by $\tilde{O}(m)$.

The reassignment process poses an additional challenge. For example, can one decide whether $u \in S_{u'}$ in time that is proportional to the number of edges (rather than nodes) in the cut $S_{u'}$ (more precisely, the reported cut between p and u' in $G_{V'}$)? Our solution utilizes an efficient dynamic-connectivity algorithm (we use a simple modification of [HK95], see Section 5.2.4), that preprocesses a graph in near-linear time, and support edge updates

[9]A random pivot was previously used in [BCH$^+$08] in the special case of an unweighted graph G, and their proof relies heavily on this restriction. Moreover, the cuts S_u in their algorithm form a laminar family, hence their reassignment process is straightforward.

and connectivity queries in polylogarithmic time — we simply delete the edges of the cut S_v and then ask if u and u' are connected.

5.2.2 Approximate Min-Cut Queries and Flow-Equivalent Trees

In this section we present our results for using *approximate* Min-Cut queries that were presented in Section 5.1 and a technical overview for them was given in Section 5.2.1.

We prove the following theorems, which formalize Informal Theorem 3 and give Corollaries 5.1.2 and 5.1.3 from Section 5.1.

Theorem 5.2.1. *There is a randomized algorithm such that given a capacitated graph $G = (V, E, c)$ on n nodes, m edges, and using $\tilde{O}(n)$ queries to a deterministic $(1+\varepsilon)$-approximate Min-Cut data structure for G with a running time t_p and amortized time t_{mc}, can with high probability:*

- *construct in time $O(t_p(n)) + \tilde{O}(n^2)$ a $(1+\varepsilon)$-approximate flow-equivalent tree T of G, and*
- *construct in time $O(t_p(n)) + \tilde{O}(n^2)$ a data structure D of size $\tilde{O}(n^2)$ that stores a set $\mathcal{C}$ of $\tilde{O}(n)$ cuts, such that given a queried pair $s, t \in V$ returns in time $\tilde{O}(1)$ a pointer to a cut in $\mathcal{C}$ that is a $(1+\varepsilon)$-approximate minimum st-cut.*

While the significance of the first item of the theorem is clear (the flow-equivalent tree) let us say a few words about why the second item is interesting compared to the assumption. The first benefit of our data structure is that it only stores $\tilde{O}(n)$ cuts and therefore it will only have $\tilde{O}(n)$ different answers to the $\binom{n}{2}$ possible queries it can receive. This makes it more similar to a cut-equivalent tree. Second, the space complexity of our data structure is upper bounded by $\tilde{O}(n^2)$ in weighted or $\tilde{O}(m)$ in unweighted graphs (see Section 5.2.4), while the oracle could have used larger space; thus we could save space without incurring loss to the preprocessing and query times by more than log factors. The third benefit is that it only uses $\tilde{O}(n)$ queries to the assumed oracle, which allows us to obtain consequences even from an oracle with larger query times and even from *offline* algorithms. If rather than a $(1+\varepsilon)$ Min-Cut data structure we have an offline $(1+\varepsilon)$-approximate minimum *st*-cut algorithm such as [KLOS14], by simply computing it every time there is a query, we get the following theorem.

Theorem 5.2.2. *If in Theorem 5.2.1 instead of a $(1+\varepsilon)$-approximate Min-Cut data structure we have an offline $(1+\varepsilon)$-approximation algorithm with running time $t_{offline}(m)$, the time bounds for constructing P and D become $\tilde{O}(n \cdot t_{offline}(n))$.*

We also remark that the above theorems only deal with deterministic data structures and algorithms. The reason will be clarified during the proof. However, this restriction can be removed and we explain how to generalize the theorem to randomized ones in Section 5.2.5.

To conclude Corollaries 5.1.2 and 5.1.3 from Section 5.1, given a graph we begin by applying a sparsification due to Benczur and Karger [BK15a], where a near-linear-time construction transforms any graph on n nodes into an $O(n \log n/\varepsilon^2)$-edge graph on the same set of nodes whose cuts $(1+\varepsilon)$-approximate the values in the original graph. This incurs a $(1+\varepsilon)$ approximation factor to the result. By utilizing a $(1+\varepsilon)$-approximate minimum *st*-cut algorithm for general capacities by [KLOS14] with $t_{\text{offline}}(m) = m^{1+o(1)}/\varepsilon^2$ we get the $n^{2+o(1)}/\varepsilon^4$ upper bound for constructing $(1+\varepsilon)$-approximate flow-equivalent trees and the tree-like data structure. The main previously known method for constructing a data structure that can answer $(1+\varepsilon)$-approximate minimum *st*-cuts is to construct an exact cut equivalent tree of a sparsification of the input graph using, e.g., Benczur-Karger [BK15b]. For general capacities, this gives a total running time of $\tilde{O}(n^{5/2})$. For unit-capacities, since this sparsification introduces edge weights, it is not clear how to do anything better for the approximation version than the exact bounds.

In the unit-capacity case, using the same techniques as in Theorem 5.2.1 (but with extra care), our bounds are better: we replace the $\tilde{O}(n^2)$ term with $\tilde{O}(m)$. While we do not currently have an application for this improved bound, it will be significant in the likely event that a $(1+\varepsilon)$-approximate Min-Cut data structure can be designed for sparse unweighted graphs that will have near-linear or even $O(n^{1.5-\delta})$ preprocessing time. Then, our improved theorem would give an approximate flow-equivalent tree construction that improves on the $n^{1.5}$ barrier that currently exists for exact [AKT20b]. We remark that, since the results of this section do not use any edge contractions and only ask queries about the original graph, they hold for *any* graph family even if it is not minor-closed. This is important since the family of sparse graphs is not minor closed. This is discussed in Section 5.2.4.

5.2.3 Our Tree-Like Data Structure

We start by proving the second item in Theorem 5.2.1 and then show how it gives the construction of approximate flow-equivalent tree in a simple way.

Let G be the input graph with node set V, we will show how to construct a data structure D that utilizes a tree structure T, and we will also construct a graph H which we will call *flow-emulator* on the same node set V that will only be used for our flow-equivalent tree construction. We assume we are given an arbitrary data structure for answering $(1+\varepsilon)$-approximate Min-Cut queries, and give a new data structure or flow-equivalent tree with error $(1+\varepsilon)^2$. Thus, to get the theorem we could use a data structure with parameter $\varepsilon' = \varepsilon/3$.

Preprocessing To construct our data structure we recursively perform *expansion* operations. Each such operation takes a subset $V' \subset V$ and partitions it into a few sets $S_i \subseteq V'$ on which the operation will be applied recursively until they have size 1 (V' can be thought of as a super-node as in Gomory-Hu but here we do not have auxiliary graphs and contractions). The partition S_i will (almost) satisfy the strong property (*) that we discussed in Section 5.2.1. In the beginning we apply the expansion on $V' := V$. It will be helpful to maintain the recursion-tree T that has a node $t_{V'}$ for each expansion operation that stores V' as well as some auxiliary information such as cuts and a mapping from each node $v \in V'$ to a cut $S_{f(v)}$. To perform a query on a pair u, v we will go to the recursion-node in T that separated them, i.e. the last V' that contains both of them, and we will return one of the cuts stored in that node.

We will prove that, because of how we build the partition, the depth of the recursion will be $O(\log n)$. For each level of the recursion, the expansion operations are performed on disjoint subsets V'_i. All the work that goes into the expansion operations in one level can be done in $O(n^2)$ time in a straightforward way. In unweighted graphs, it can even be done in $\tilde{O}(m)$ time by adapting known dynamic connectivity algorithms; this will be discussed in Section 5.2.4.

The expansion operation on a subset $V' \subseteq V$ (it is helpful to think of the case $V' = V$):

1. Pick a pivot node $p \in V'$ uniformly at random.

2. For every node $u \in V' \setminus \{p\}$ ask a $(1+\varepsilon)$-approximate Min-Cut query for the pair u, p to get a cut $(V \setminus S_u, S_u)$ where $u \in S_u$ and $p \in V \setminus S_u$. Compute the value of the cut and denote it by $c(S_u)$. Moreover, compute the intersection of the side of u with V', that is $S_u \cap V'$, and denote this set by S'_u.

3. Treat the cut values as being all different by breaking ties arbitrarily and consistently. One way is to redefine the value $c(S)$ of the cut S to be $c(S) + i/n^2$ if S was the answer to the i^{th} Min-Cut query we performed. From now on assume that all $c(S)$ values are unique.

4. We would like to use the sets S'_u for each $u \in V'$ to partition V', but these sets can be intersecting in arbitrary ways and moving nodes around could hurt our property (*). The following is a carefully designed *reassignment* process that makes it work. There are three main criteria when reassigning nodes to cuts. First, we can only assign a node v to a cut S_u whose value is within $(1+\varepsilon)$ of the best cut separating v and p; this is necessary to satisfy property (*). Second, we want to prioritize assigning v to a cut S_u separating it from p with good value that also has *small cardinality* S'_u; this will make sure the sets are getting smaller with each recursive step and upper bound the depth of the recursion by $O(\log n)$. And third, a subtle but crucial criterion for satisfying property (*) is that we may not assign two nodes u, v to two different sets unless we have evidence for doing so in the form of a cut S with good value that separates one but not the other from p (and therefore separates them). While each of these criteria is easy to satisfy on its own, getting all of them requires the following complicated process.

   We define a reassignment function $f : V' \to V' \cup \{\perp\}$ such that for every node $u \in V' \setminus \{p\}$ with cut $(V \setminus S_u, S_u)$, we reassign u to v, denoted $f(u) = v$ with the cut $(V \setminus S_{f(u)}, S_{f(u)})$ as follows. Denote by V_{small}, V'_{small} two initially identical sets, each containing all nodes u such that $|S'_u| \leq n'/2$, where $|V'| = n'$, and denote by $V_{big}, V'_{big}, V''_{big}$ three sets that are initially all equal to $V' \setminus V_{small}$. As a preparation for defining f we need another function g that reassigns nodes in V_{big} to the best cut corresponding to another node in V_{big} that separates them from p. Sort V_{big} by $c(S_u)$, and for all $u \in V_{big}$ from low $c(S_u)$ to high and for every node $v \in S'_u \cap V'_{big}$, set $g(v) = u$ and then remove v from V'_{big}. Sort V_{small} by $c(S_u)$, and for all $u \in V_{small}$ from low $c(S_u)$ to high and for every node $v \in S'_u \cap V'_{small}$, set $f(v) = u$ and then remove v from V'_{small}. For every node $v \in S'_u \cap V''_{big}$, if $c(S_u) \leq (1+\varepsilon)c(S_{g(v)})$ then set $f(v) = u$ and then remove v from V''_{big}. Finally, set $f(v) = \perp$ for every node v for which f was not assigned a value (including p).

   To get the partition, let $IM(f)$ be the image of f (excluding $\perp$) and for each $i \in IM(f)$ let $f^{-1}(i)$ be the set of all nodes u that were reassigned by f to the cut $S_{f(i)}$. Notice that the nodes in V''_{big}, which includes p, were not assigned to any set. Thus, we get the partition of V' into V''_{big} and each set in $\{f^{-1}(i)\}_{i \in IM(f)}$. The latter sets satisfy the property (*) but V''_{big} may not (because it does not correspond to an approximate minimum cut) and therefore it will be handled separately next.

5. If $|V_{small}| < n'/4$ then p is a failed pivot. In this case, re-start the expansion operation at step 1 and continue to choose new pivots until $|V_{small}| \geq n'/4$. We will prove that we will only do $O(\log n)$ repetitions with high probability.

6. Finally, we recursively compute the expansion operation on each of the sets of the partition. Let us describe what we store at the recursion node $t_{V'}$ corresponding to the just-completed expansion operation on V' with (successful) pivot p. Simultaneously, we describe what we add to the flow-emulator graph H (that will be used in for constructing a flow-equivalent tree in unweighted graphs more efficiently in Section 5.2.4) which initially has no edges, but gets $|V'| - 1$ new weighted edges with each expansion operation. If $|V'| = 1$ we do nothing, so assume that $|V'| \geq 2$. We store $|V'| - 1$ cuts in $t_{V'}$: For each node $v \in V''_{big}$ that is not p we store the cut $S_{g(v)}$ and we also add an edge between p and v in the flow-emulator graph H with weight $(1+\varepsilon)c(S_{g(v)})$. And for each node u in one of the other sets of the partition $\{f^{-1}(i)\}_{i \in IM(f)}$ we store the cut it was reassigned to $S_{f(u)}$ and we also add an edge $\{v, p\}$ of weight $(1+\varepsilon)c(S_{f(u)})$ to H. If any of these edges already exists in H (which could happen for the nodes $v \in V''_{big}$) then we simply do nothing and keep the previous edge. We also keep an array of pointers from each node to its corresponding cut and also the value of the cut, call this array A. Moreover, we store for each node of V' the name of the set in the partition that it belongs to, in an array B.

Queries To answer a query for a pair u, v we go to the recursion level that separated them, corresponding to some node $t_{V'}$ in T and output a pointer to one of the two corresponding cuts S_u or S_v; choose the cut among the two that separates u and v (we prove that at least one of the two cuts does) and has smaller capacity. To find out which recursive node separates u and v we can simply start from the root and continue going down (with the help of array B) to the nodes that contain both of them until we reach V'. The query time will depend on the depth of the recursion which we will show to be logarithmic.

Correctness The next claim proves that the cuts our data structure returns are approximately optimal. The main idea is to prove that the partition we get at each expansion step satisfies the property (*) discussed in Section 5.2.1, except for the set V''_{big} which has to be treated separately; things work out because there is only one such problematic set.

Claim 5.2.3. *The cut returned by D for any pair of nodes is a $(1+\varepsilon)^2$ approximate minimum cut. Moreover, for any pair $u, v \in V$ there exists a special node $p_{uv} \in V$ such that*

$$(1+\varepsilon)^3 \textsf{\textit{Max-Flow}}(u, v) \geq \min\{c_H(u, p_{uv}), c_H(v, p_{uv})\} \geq \textsf{\textit{Max-Flow}}(u, v),$$

where c_H is the weight of the edge in our flow-emulator graph H.

Proof. Let u, v be an arbitrary pair of nodes and let $V' \subseteq V$ be the set such that $u, v \in V'$ but u and v were sent to different sets in the expansion operation on V' during the construction of D. There are a few cases, depending on whether any of them is in V''_{big} or not, and whether the cuts they got assigned to had similar costs up to $(1+\varepsilon)$.

1. The first case is when none of u, v are in V''_{big}. Assume without loss of generality that $c(S_{f(u)}) > c(S_{f(v)})$ where $S_{f(u)}$ and $S_{f(v)}$ are the corresponding cuts. There are two sub-cases, depending on whether the values of the two cuts are close or not.

(a) If $c(S_{f(u)}) > (1+\varepsilon)c(S_{f(v)})$ then

$$\mathsf{Max\text{-}Flow\text{-}Value}(u,p) > \mathsf{Max\text{-}Flow\text{-}Value}(v,p),$$

and so

$$\mathsf{Max\text{-}Flow\text{-}Value}(v,u) = \mathsf{Max\text{-}Flow\text{-}Value}(v,p).$$

As a result, it must be that $u \in V' \setminus S'_{f(v)}$ and $S_{f(v)}$ is indeed the cut returned, with $(1+\varepsilon)$ approximation ratio.

(b) Otherwise, if $c(S_{f(u)}) \leq (1+\varepsilon)c(S_{f(v)})$ then it must be that $u \in V' \setminus S'_{f(v)}$, since otherwise when the algorithm examined $f(v)$, it was the case that both u and v were in $S'_{f(v)}$, and as they are in V_{small} they must had been sent to the same recursion instance, contradicting our assumption on the expansion operation on V', and so

$$\mathsf{Max\text{-}Flow\text{-}Value}(u,v) \leq c(S_{f(v)}).$$

Furthermore,

$$\mathsf{Max\text{-}Flow\text{-}Value}(u,v) \geq \min(\mathsf{Max\text{-}Flow\text{-}Value}(u,p), \mathsf{Max\text{-}Flow\text{-}Value}(v,p))$$

and thus

$$(1+\varepsilon)\mathsf{Max\text{-}Flow\text{-}Value}(u,v) \geq \min(c(S_{f(u)}), c(S_{f(v)})).$$

By our assumption, $c(S_{f(u)}) > c(S_{f(v)})$ and so altogether

$$(1+\varepsilon)\mathsf{Max\text{-}Flow\text{-}Value}(u,v) \geq c(S_{f(v)}).$$

Thus, the algorithm can output $S_{f(v)}$ with an approximation guarantee $(1+\varepsilon)$, as required.

2. The second case is when one of the nodes is in V''_{big} and its Max-Flow to p is larger. More specifically, let $u_{big} \in V''_{big}$ and $v \notin V''_{big}$ be nodes such that $c(S_{g(u_{big})}) > c(S_{f(v)})$, where $S_{g(u_{big})}$ is the cut corresponding to u_{big}. Again, there are two sub-cases.

   (a) If $c(S_{g(u_{big})}) > (1+\varepsilon)c(S_{f(v)})$ then similar to before, $S_{f(v)}$ separates u_{big} and v, providing a $(1+\varepsilon)$-approximation.

   (b) Otherwise, if $c(S_{g(u_{big})}) \leq (1+\varepsilon)c(S_{f(v)})$ then it must be that $u_{big} \in V' \setminus S'_{f(v)}$, since if not then as $u_{big} \in V_{big}$ and when the algorithm examined $f(v)$ it did not set $f(u_{big}) := f(v)$, it must have been the case for a node x that was either $f(v)$ or before $f(v)$ in the order (i.e. such that $c(S_x) \leq c(S_{f(v)})$) that u_{big} was tested for the first time, with $c(S_x) > (1+\varepsilon)c(S_{g(u_{big})})$, and so $c(S_{f(v)}) > (1+\varepsilon)c(S_{g(u_{big})})$. However, by our assumption it holds that $c(S_{f(v)}) < c(S_{g(u_{big})})$, in contradiction. Thus, $u_{big} \in V' \setminus S'_{f(v)}$. Similar to before,

   $$(1+\varepsilon)\mathsf{Max\text{-}Flow\text{-}Value}(u_{big}, v) \geq c(S_{f(v)}),$$

   and thus the returned cut $S_{f(v)}$ is a $(1+\varepsilon)$ approximation, as required.

3. The third and last case is when one of the nodes is in V''_{big} and its Max-Flow to p is smaller. Let $u_{big} \in V''_{big}$ and $v \notin V''_{big}$ be nodes such that $c(S_{f(v)}) > c(S_{g(u_{big})})$. There are two sub-cases.

(a) If $c(S_{f(v)}) > (1+\varepsilon)c(S_{g(u_{big})})$ then similar to before, $S_{g(u_{big})}$ separates u_{big} and v, providing a $(1+\varepsilon)$-approximation.

(b) Otherwise, if $c(S_{f(v)}) \le (1+\varepsilon)c(S_{g(u_{big})})$ then it must be that $u_{big} \in V' \setminus S'_{f(v)}$. Otherwise, since $u_{big} \in V_{big}$ and when the algorithm examined $f(v)$ it did not set $f(u_{big}) := f(v)$, it must have been the case that $c(S_{f(v)}) > (1+\varepsilon)c(S_{g(u_{big})})$. However, by our assumption it holds that $c(S_{f(v)}) \le (1+\varepsilon)c(S_{g(u_{big})})$, in contradiction. Thus, $u_{big} \in V' \setminus S'_{f(v)}$. By previous arguments,

$$(1+\varepsilon)\textsf{Max-Flow-Value}(u_{big}, v) \ge \min\{c(S_{big}), c(S_{f(v)})\},$$

and since $1/(1+\varepsilon)c(S_{f(v)}) \le c(S_{g(u_{big})})$, it must be that

$$(1+\varepsilon)\textsf{Max-Flow-Value}(u_{big}, v) \ge 1/(1+\varepsilon)c(S_{f(v)}),$$

and finally

$$(1+\varepsilon)^2 \cdot \textsf{Max-Flow-Value}(u_{big}, v) \ge S_{f(v)}$$

providing an approximation ratio of $(1+\varepsilon)^2$, concluding the claim.

To prove the statement about the weights in H simply observe that the weights in H correspond exactly to $(1+\varepsilon)$ times the weights of the cuts that were considered in the proof above. Note that when p_{uv} is the pivot separating u and v, i.e., the pivot that sent u and v to different instances in an expansion step, it might be the case that the returned cut's capacity is the bigger out of the cuts of (u, p_{uv}) and (v, p_{uv}), in particular it happens in case 3b in the above proof. However, in this case the smaller value is at least $1/(1+\varepsilon)$ times the bigger value, and so the fact that we multiplied all values by $(1+\varepsilon)$ when we added them to H on one hand ensures the lower bound of $\textsf{Max-Flow}(u, v)$ and on the other hand increases the upper bound by a factor of $(1+\varepsilon)$ to be concluded as $(1+\varepsilon)^3\textsf{Max-Flow}(u, v)$.

□

Running Time Next we prove the upper bounds on the preprocessing time, by proving that with high probability, the algorithm terminates after $\tilde{O}(n^2)$ time. The crux of the argument is to bound the depth of the recursion by $O(\log n)$. Later, in Section 5.2.4 we build on this analysis to show that our more efficient implementation for unweighted graphs gives an upper bound of $\tilde{O}(m)$. There, we show that a single expansion step takes only $\tilde{O}(m)$ rather than $O(n^2)$ but the rest of the analysis is the same.

Let us give a high-level explanation of the argument below. Our goal is to bound the size of each of the sets in the partition in an expansion operation by $3/4|V'|$. This is immediate for the sets $\{f^{-1}(i)\}_i$ because they are subsets of cuts S_u of nodes u in V_{small}, and by definition they satisfy that $|S_u| \le |V'|/2$. Therefore, we should only worry about V''_{big}. However, any node u that is initially in V_{small} will end up reassigned to one of the sets $\{f^{-1}(i)\}_i$ and not to V''_{big}. Thus, it suffices to argue that there will be at least $|V'|/4$ nodes in V_{small}. To argue about this, let us recall where the cuts S_u for each node u come from. They are the approximate Min-Cuts that our assumed data structure returns when queried for pairs u, p for a randomly chosen pivot p. For simplicity, let us assume that this data structure is deterministic (we show how to lift this assumption in Section 5.2.5) which means that for any pair x, y the answer to the query will always be a certain cut (S_x, S_y) and in this cut it must be that either $|S_x| \le n/2$ or $|S_y| \le n/2$ or both. (More generally, if we take the intersection of each side of the cut with a subset $V' \subseteq V$ we can replace $n/2$ by $|V'|/2$, as we will do below.) Therefore, the u, p query has a chance of at least $1/2$ of having $|S_u| \leq |V'|/2$ meaning that u is in V_{small}. To complete the argument, we need a stronger property: we want that for a randomly chosen p, at least $1/4$ of the nodes $u \in V'$ will have that the side of u is smaller than the side of p and they will end up in V_{small}. This is argued more formally below.

We start with a general lemma about tournaments.

Lemma 5.2.4. *Let $Y = (V_Y, E_Y)$ be a directed graph on n nodes and m edges that contains a tournament on V_Y. Then Y contains at least $n/2$ nodes with out-degree at least $n/4$.*

Proof. Each edge contributes exactly 1 to the total sum of the out-degrees and the in-degrees. Thus, these two sums are equal and so the average out-degree in Y equals $\sum_{v \in V_Y} outdeg_Y(v)/n = m/n \geq \binom{n}{2}/n = (n-1)/2$. Using the probabilistic method, we get that there exists a node with out-degree that is at least $(n-1)/2$. By removing this node and using similar arguments repeatedly, we conclude that there exist $\lceil n/2 \rceil$ nodes with degrees at least $(n-1)/2, (n-2)/2, \ldots, (n - \lceil n/2 \rceil)/2$, i.e. at least $n/4$. □

The following is a general corollary, and is a result of Lemma 5.2.4, about cuts between every pair of nodes.

Corollary 5.2.5. *Let $F = (V_F, E_F)$ be a graph where each pair of nodes $u, v \in V_F$ is associated with a cut $(S_{uv}, S_{vu} = V_F \setminus S_{uv})$ where $u \in S_{uv}, v \in S_{vu}$ (possibly more than one pair of nodes are associated with each cut), and let $V'_F \subseteq V_F$. Then there exist $|V'_F|/2$ nodes p' in V'_F such that at least $|V'_F|/4$ of the other nodes $w \in V'_F \backslash \{p'\}$ satisfy $|S_{p'w} \cap V'_F| > |S_{wp'} \cap V'_F|$.*

Proof. Let $H_F(V'_F)$ denote the *helper graph* of F on V'_F, where there is a directed edge from $u \in V'_F$ to $v \in V'_F$ if and only if $|S_{uv} \cap V'_F| > |S_{vu} \cap V'_F|$. By Lemma 5.2.4, since $H_F(V'_F)$ contains a tournament on V'_F, Corollary 5.2.5 holds. □

Next, apply Corollary 5.2.5 on G, and let $H = H_G(V')$ be the helper graph of G on V' with the reassigned cuts. As a result, with probability at least $1/2$, the pivot p is one of the nodes with out-degree at least $n'/4$, and in that case, when the algorithm partitions V', it must be that $\max_i |f^{-1}(i)| \leq n'/4$, and $|f^{-1}(\perp)| \leq 3n'/4$, that is, the largest set created is of size at most $3n'/4$. After $O(n \log n)$ successful choices of p, the algorithm finishes with the total depth of the recursion being $O(\log_{4/3} n)$. Note that the algorithm verifies the choice of p and never proceeds with an unsuccessful one. Hence, it is enough to bound the running time of the algorithm given only successful choices of p by $\tilde{O}(n^2)$ and $\tilde{O}(m)$ in the general case and in the unit edge-capacities case, respectively, and then multiply by the maximal number of unsuccessful choices for any instance, which is bounded by $3 \log n$ with high probability, as shown below.

A straightforward implementation of an expansion step gives an upper bound of $\tilde{O}(n^2)$ on the total running time for the algorithm given only successful choices of p. In Lemma 5.2.8 we prove the better upper bound of $\tilde{O}(m)$ for unweighted graphs.

Finally, the probability for failure of $3 \log n$ consecutive trials in a single instance is at most $(1/2)^{3 \log n} = 1/n^3$, and by the union bound over the $\tilde{O}(n)$ instances in the recursion, the probability that at least one instance takes more than $3 \log n$ attempts to have a successful choice of p is bounded by $1/n$. We conclude that with high probability, the running time of the algorithm is bounded by $O(t_p(n)) + \tilde{O}(n^2)$ for general capacities and $O(t_p(m)) + \tilde{O}(m)$ for unit edge-capacities, as required.

Space Usage In the general weighted case, the total space usage is $\tilde{O}(n^2)$: There are $O(\log n)$ levels and in each level the expansion operations are performed disjoint sets V'. Each operation stores arrays of size $|V'|$, containing pointers, values, and cuts. Each cut can take $O(m)$ bits, but since we can apply the Benczur-Karger sparsification we can assume that $m = \tilde{O}(n)$ (unless we are in the unweighted setting which we will discuss separately). Therefore, the total size at each recursive level is $\tilde{O}(n^2)$ and we are done. In unweighted graphs, we will argue in Section 5.2.4 that for any partition of V and any choices of pivots in each of the parts, the total number of edges in all minimum cuts from the pivots to the nodes in their parts is upper bounded by $O(m)$. The fact that we are dealing with approximations only incurs a $(1+\varepsilon)$ factor to this cost. Therefore, we can store all the cuts in a single recursive level in $O(m)$ space, and the other arrays only take $O(n \log n)$ space per level. In total, we get the $\tilde{O}(m)$ bound.

Flow-Equivalent Tree Construction We apply a technique of Gomory and Hu [GH61]. Our data structure lets us to query for the approximate Max-Flow value for a pair of nodes in $\tilde{O}(1)$ time. We have the following proposition, extending the technique of [GH61] to approximated values of an input graph G.

Proposition 5.2.6. *Let $G = (V, E)$ be an input graph and $N = (V, c)$ a complete graph on V such that for every two nodes $u, v \in V$, $(1+\varepsilon)\textsf{Max-Flow}(u,v) \geq c_N(u,v) \geq \textsf{Max-Flow}(u,v)$. Then a maximum spanning tree T of N is a $(1+\varepsilon)$-approximate flow-equivalent tree of G.*

Proof. To prove the claim about T, let u, v be any two nodes and consider any uv-path in T $u_1 = u, \ldots, u_k = v$, and we will show that

$$(1+\varepsilon)\textsf{Max-Flow}(u,v) \geq \min\{c_N(u_1,u_2), \ldots, c_N(u_{k-1},u_k)\} \geq \textsf{Max-Flow}(u,v).$$

For the first inequality, we follow the original proof for the exact case [GH61], where it is shown that for any path $u_1 = u, \ldots, u_k = v$ in the complete network representing exact answers, it holds that

$$\textsf{Max-Flow}(u_1,u_k) \geq \min\{\textsf{Max-Flow}(u_1,u_2), \ldots, \textsf{Max-Flow}(u_{k-1},u_k)\}.$$

This is proved by induction. By the strong triangle inequality

$$\textsf{Max-Flow}(u_1,u_k) \geq \min\{\textsf{Max-Flow}(u_1,u_{k-1}), \textsf{Max-Flow}(u_{k-1},u_k)\},$$

and by the inductive hypothesis

$$\textsf{Max-Flow}(u_1,u_{k-1}) \geq \min\{\textsf{Max-Flow}(u_1,u_2), \ldots, \textsf{Max-Flow}(u_{k-2},u_{k-1})\}.$$

Thus, in our approximate setting and by our construction, it must follow that

$$\textsf{Max-Flow}(u,v) \geq 1/(1+\varepsilon) \min\{c_N(u_1,u_2), \ldots, c_N(u_{k-1},u_k)\}.$$

The second inequality relies on the properties of any path in a maximum-weight spanning tree, as follows. For any path $u_1 = u, \ldots, u_k = v$ between u and v in T it holds that

$$\min\{c_N(u_1,u_2), \ldots, c_N(u_{k-1},u_k)\} \geq c_N(u,v).$$

Indeed, otherwise the edge uv must not be in T, and it could thus replace the minimum-weight edge in the path $u_1, \ldots, u_k$ in T while increasing the total weight of the edges in T, in contradiction.

□

This allows us to construct, in $\tilde{O}(n^2)$ time, a complete graph N on V that has an edge of weight $c_N(s,t)$ between any pair of nodes s,t such that $(1+\varepsilon)^2\mathsf{Max\text{-}Flow}(s,t) \geq w(s,t) \geq \mathsf{Max\text{-}Flow}(s,t)$. By Proposition 5.2.6, the maximum spanning tree (MST) of this complete graph is a $(1+\varepsilon)^2$-approximate flow-equivalent tree of G.

5.2.4 A Faster Implementation For Unweighted Graphs

In this section we explain how to improve the bounds of Theorem 5.2.1 in the case of unweighted graphs.

Theorem 5.2.7. *For graphs $G = (V,E)$ with unit edge-capacities, the time bounds in Theorem 5.2.1 for constructing T and D become $t_p(m) + \tilde{O}(m)$, and the space bound for D becomes $\tilde{O}(m)$.*

First, we show that an expansion step can be executed more efficiently in unweighted graphs by only spending time proportional to the number of edges in all the cuts we process. In unweighted graphs the total size is only $O(m)$. This is challenging because our reassignment needs to analyze which nodes are in each cut and what is the best value for each one. We have managed to do this by adapting known data structures for dynamic graph connectivity.

Lemma 5.2.8. *The running time for the algorithm given only successful choices of p is bounded by $\tilde{O}(m)$ for graphs with unit edge-capacities.*

Proof. For unit edge-capacities, we first show that the total space of all cuts examined by the algorithm is bounded by $\tilde{O}(m)$, and then that the running time is linear in that measure. Indeed, the cuts computed in each recursion depth are between pivot-sink pairs such that a pivot in one instance is never a sink in another instance in the same depth. Let $Q_i \subseteq V \times V$ denote the set containing all pairs of nodes queried in depth i. Denote by T a cut-equivalent tree of G, and by α_T the (multi-)set of edges in T that are the answers to (exact) $\mathsf{Min\text{-}Cut}$ queries in T of the pairs in Q_i. We assume that for every pair t,p in Q_i, the edge in T answered is the one touching t. Note that our assumption could have only increased the total capacity of the edges in α_T. Since no node can be both a pivot and a sink in the same depth, it must be that every edge in T is returned and added to α_T at most twice, and since the sum of all edge-capacities in T is $2m$ (see Lemma 5 in [BHKP07]), an $O(m)$ bound for the total capacity of the edges in α_T follows. Since the capacity of every edge in T is the number of edges in the cut it represents, and the cuts our algorithm uses are $(1+\varepsilon)$-approximated, they contain at most $(1+\varepsilon)$ times the number of edges in the cuts corresponding to the edges in α_T, as claimed.

Now, to see that the running time is bounded, first note that for every cut S_u examined by the algorithm throughout its execution, nodes $v \in S'_u$ are examined and they either getting a value under g or f, or removed from the corresponding set it belonged to, V'_{big}, V_{small}, or V''_{big}, so we are left with showing that counting and reporting a set S'_u could be done in $\tilde{O}(1)$ and $O(|S'_u|)$ time, respectively. In fact, for each S_u we will consider a subset of S_u that is the connected component in $G \setminus \delta(S_u)$ containing u, where $\delta(S_u)$ is the set of edges leaving S_u, with additional running time of $O(\delta(S_u))$, and $\tilde{O}(m)$ for all cuts S_u's. We explain these steps below.

Claim 5.2.9. *Let $G = (V,E)$ be a graph and $V_T \subseteq V$ a subset of terminals. For every cut $S \subseteq V$ given by the edges $\delta(S)$ and every node $y \in S$, it is possible to count the nodes in*

$S(y) \cap V_T$ for a cut $S(y) \subseteq S$ that is the connected component of $G[S]$ that contains y, in time $O(|\delta(S)|)$, and enumerate $S(y) \cap V_T$ in additional time $O(|S(y) \cap V_T|)$.

Proof. The idea is to slightly modify a known dynamic connectivity algorithm [HK95], as follows. In [HK95], by using Euler Tour Trees (ETTs) implemented by Binary Search Trees (BSTs) a dynamic forest is maintained, each of whose trees representing a connected component in the graph. The important feature of ETTs we utilize here is that their BST implementation is well suited for storing and answering aggregate information on its subtrees, in addition to supporting elementary operations such as finding the root of a tree containing a node, cutting and linking a subtree from and to trees, and answering if two nodes are connected, all in $\tilde{O}(1)$ time. Thus, the information we keep for every subtree is the size of its intersection with V_T. Next, using the dynamic algorithm, remove the edges $\delta(S)$, denoting the resulting graph by G_S and the connected component of y in G_S by C_y. Then enumerate every edge in the cut $\delta(S)$ and remove every edge that neither of its ends lies in C_y, resulting in a cut $S(y) = C_y$ containing y and such that $c(S(y)) \leq c(S)$, as in the claim. In order to report $S(y) \cap V_T$, simply output the aggregated information in the root of the BST corresponding to $S(y)$. To enumerate the nodes in $S(y) \cap V_T$, traverse the BST of the connected component $S(y)$ starting with the root, and follow a child whose intersection with V_T is ≥ 1, until arriving at a leaf which is then enumerated. The total time spent for removing the cut edges and reporting the intersection size is thus $O(|\delta(S)|)$, and an additional time of $O(|S(y) \cap V_T|)$ is spent on traversing the BST and enumerating the nodes in $S(y) \cap V_T$. $\square$

We use claim 5.2.9 on our instance by first preprocessing the cuts S_y the algorithm computed and switch them with the corresponding cuts $S_y(y)$ in total time $\tilde{O}(m)$ for the current depth (as shown in the beginning of this proof), and then setting V_T to be either V', V'_{big}, V_{small}, or V''_{big}, which incurs an addition of $O(|V'|+|V'_{big}|+|V_{small}|+|V''_{big}|) = O(V')$ to the running time, bringing the total running time at a single depth to $\tilde{O}(m)$, as required. Multiplying by the height of the recursion, which is at most $O(\log_{4/3} n)$, concludes the proof. $\square$

As claimed before, there are at most $\tilde{O}(1)$ unsuccessful choices of pivots per a successful one, thus the total time for constructing D is $\tilde{O}(m)$, as required.

Flow-Equivalent Tree Construction for Unweighted Graphs We use the flow-emulator H to compute a flow-equivalent tree without spending $\Omega(n^2)$ time as in the general case.

Lemma 5.2.10. *A flow equivalent tree T can be constructed from H in near linear time in the size of H, such that T represents a $(1+\varepsilon)^3$ approximation of the correct* Max-Flow *values.*

Proof. The algorithm is to simply pick a maximum spanning tree T_H of the flow-emulator H. In order to prove that T_H is an approximate flow-equivalent tree of the input graph G, consider a complete graph H' on V that is constructed from H by adding an edge between every pair of nodes u, v that did not have an edge in H, with capacity $c(uv) = \min\{c_H(u, p_{uv}), c_H(v, p_{uv})\}$, for the special node p_{uv} from Claim 5.2.3. This claim and the construction of H' imply that for every pair uv in H',

$$(1+\varepsilon)^3 \text{Max-Flow}_G(u,v) \geq c_{H'}(u,v) \geq \text{Max-Flow}_G(u,v).$$

We show that there exists a maximum spanning tree of H' that does not pick the newly added edges. It will follow that T_H is also a maximum spanning tree of H' and thus, by Proposition 5.2.6, T_H is a $(1+\varepsilon)^3$-approximate flow-equivalent tree of G, as required.

Now, let $T_{H'}$ be any maximum spanning tree of H'. In what follows we show that new edges could always be replaced by edges from H in a way that does not decrease the weight of $T_{H'}$. We call an edge uv in $T_{H'}$ a new edge if it does not exist in H. For every new edge uv in $T_{H'}$ that satisfies, without loss of generality, that $c_{H'}(p_{uv}, u) \geq c_{H'}(p_{uv}, v)$ (the case $c_{H'}(p_{uv}, u) \leq c_{H'}(p_{uv}, v)$ is symmetric), replace uv with an edge in H according to the first of the following rules that applies (note that at least one must be true).

1. If the edge $p_{uv}v$ is in $T_{H'}$, then replacing uv with $p_{uv}u \in E(H)$ could only increase the weight of $T_{H'}$.

2. If the edge $p_{uv}u$ is in $T_{H'}$, then replacing uv with $p_{uv}v \in E(H)$ would keep the weight of $T_{H'}$ the same.

3. If neither of the edges $p_{uv}u$ and $p_{uv}v$ is in $T_{H'}$, then

    (a) If the path in $T_{H'}$ between p_{uv} and v, denoted P'_{pv}, does not contain the edge uv, then we replace uv with $p_{uv}u \in E(H)$, which could only increase the total weight of the tree.

    (b) If P'_{pv} does contain the edge uv, then we replace uv with $p_{uv}v \in E(H)$, keeping the total weight of the tree the same.

At the end, $T_{H'}$ remains only with edges that are in H. Thus, we concluded Lemma 5.2.10. □

5.2.5 Handling Randomized Data Structures

To bound the depth of the recursion by $O(\log n)$ we argued (using Lemma 5.2.4 about tournaments) that for a randomly chosen pivot p it will be the case that for at least a $1/4$ of the targets u the side of u in the cut *returned by our hypothetical data structure* is smaller. If the data structure we wish to use is randomized, there could be an issue because the returned cut could change each time we ask this query (or if we ask the query as (p, u) or (u, p)), and the notions we use in the arguments are not well-defined. Here we show how to avoid these issues by a more careful analysis that fixes the random bits used by the data structure.

First, for Theorem 5.2.1 we assume that the preprocessing step is deterministic and the queries are randomized, and note that it is enough to consider this case also for Theorem 5.2.2 that deals with offline $(1 + \varepsilon)$-approximate minimum *st*-cut algorithms, called henceforth $(1+\varepsilon)MinCut(s, t)$. Generate a sequence of $O(t_{\text{offline}}(m))$ random coins, and use these coins for every application of $(1 + \varepsilon)MinCut(s, t)$, keeping the results consistent in the following way. For a pair s, t queried by the algorithm, apply $(1+\varepsilon)MinCut(s, t)$ or $(1+\varepsilon)MinCut(t, s)$, according to increasing order of s and t's binary representation. By standard amplification techniques and union bound, we assume that for all pairs s, t, $(1 + \varepsilon)MinCut(s, t)$ succeed. Thus, the tournament in Lemma 5.2.4 is well-defined, and this case is concluded. Second, we assume the preprocessing step is randomized, and the queries are deterministic. In this case, by union bound over all $\binom{n}{2}$ pairs of distances, $(1 + \varepsilon)MinCut(s, t)$ succeeds. Finally, if both preprocessing and queries are randomized, generate first all random coins as described in the previous two cases, then apply union bound over the two of them.

5.3 Algorithm for a Cut-Equivalent Tree

In this section we show a new algorithm for constructing a cut-equivalent tree for graphs from a minor-closed family $\mathcal{F}$ (for example all graphs), given a Min-Cut data structure for this family $\mathcal{F}$. For ease of exposition, we first assume that the data structure supports also Max-Flow queries (reporting the value of the cut) in time $t_{mf}(m)$; we will later show that Min-Cut queries suffice.

Theorem 5.3.1. *Given a capacitated graph $G \in \mathcal{F}$ on n nodes and m edges, and access to a deterministic Min-Cut data structure for $\mathcal{F}$ with preprocessing time $t_p(\cdot)$ and output sensitive time $t_{mc}(\cdot)$, one can construct, with high probability, a cut-equivalent tree for G in time $\tilde{O}(t_p(m) + m \cdot t_{mc}(m))$. Furthermore, it suffices that the data structure's queries are restricted to a fixed source.*

By combining our algorithm with the Min-Cut data structure of Arikati, Chaudhuri, and Zaroliagis [ACZ98] for graphs with treewidth bounded by (a parameter) t, which attains $t_p = n \log n \cdot 2^{2^{O(t)}}$ and $t_{mc} = t_{mf} = 2^{2^{O(t)}}$, we immediately get the first near-linear time construction of a cut-equivalent tree for graphs with bounded treewidth, as follows.

Corollary 5.3.2 (Expanded Corollary 5.1.1)**.** *Given a graph G with n nodes and treewidth at most t, one can construct, with high probability, a cut-equivalent tree for G in time $\tilde{O}(2^{2^{O(t)}} n)$.*

The rest of this section is devoted to proving Theorem 5.3.1. Our analysis relies on the classical Gomory-Hu algorithm [GH61], hence we start by briefly reviewing it (largely following [AKT20b]) with a bit more details than in Section 5.2.1.

The Gomory-Hu algorithm. This algorithm constructs a cut-equivalent tree $\mathcal{T}$ in iterations. Initially, $\mathcal{T}$ is a single node associated with V (the node set of G), and the execution maintains the invariant that $\mathcal{T}$ is a tree; each tree node i is a *super-node*, which means that it is associated with a subset $V_i \subseteq V$; and these super-nodes form a partition $V = V_1 \sqcup \cdots \sqcup V_l$. Each iteration works as follows: pick arbitrarily two graph nodes s, t that lie in the same tree super-node i, i.e., $s \neq t \in V_i$, then construct from G an auxiliary graph G' by merging nodes that lie in the same connected component of $\mathcal{T} \setminus \{i\}$, and invoke a Max-Flow algorithm to compute in G' a minimum st-cut, denoted C'. (For example, if the current tree is a path on super-nodes $1, \ldots, l$, then G' is obtained from G by merging $V_1 \cup \cdots \cup V_{i-1}$ into one node and $V_{i+1} \cup \cdots \cup V_l$ into another node.) The submodularity of cuts ensures that this cut is also a minimum st-cut in the original graph G, and it clearly induces a partition $V_i = S \sqcup T$ with $s \in S$ and $t \in T$. The algorithm then modifies $\mathcal{T}$ by splitting super-node i into two super-nodes, one associated with S and one with T, that are connected by an edge whose weight is the value of the cut C', and further reconnecting each j which was a neighbor of i in $\mathcal{T}$ to either super-node S or T, depending on which side of the minimum st-cut C' contains V_j.

The algorithm performs these iterations until all super-nodes are singletons, and then $\mathcal{T}$ is a weighted tree with effectively the same node set as G. It is proved in [GH61] that for every $s, t \in V$, the minimum st-cut in $\mathcal{T}$, viewed as a bipartition of V, is also a minimum st-cut in G, and of the same cut value. We stress that this property holds regardless of the choices, made at each iteration, of two nodes $s \neq t \in V_i$.

5.3.1 The Algorithm for General Capacities

We turn out attention to proving Theorem 5.3.1. Let $G = (V, E, c)$ be the input graph. We shall make the following assumption, justified by a standard random-perturbation argument that we provide for completeness in Section 5.3.6.

Assumption 5.3.3. *The input graph G has a single cut-equivalent tree $\mathcal{T}^*$, with $n-1$ distinct edge weights.* [10]

5.3.2 Overview of the Algorithm

At a very high level, our algorithm accelerates the Gomory-Hu algorithm by performing every time a batch of Gomory-Hu steps instead of only one step. Similarly to the actual Gomory-Hu algorithm, our algorithm is iterative and maintains a tree $\mathcal{T}$ of super-nodes, which means that every tree node i is associated with $V_i \subseteq V$, and these super-nodes form a partition $V = V_1 \sqcup \cdots \sqcup V_l$. This tree $\mathcal{T}$ is initialized to have a single super-node corresponding to V, and since it is modified iteratively, we shall call $\mathcal{T}$ the *intermediate tree.* Eventually, every super-node is a singleton and the tree $\mathcal{T}$ corresponds to $\mathcal{T}^*$.

In a true Gomory-Hu execution, every iteration partitions some super-node i into exactly two super-nodes, say $V_i = S \sqcup T$, which are connected by an edge according to the minimum cut between a pair $s \in S, t \in T$ that is computed in an auxiliary graph. In contrast, our algorithm partitions a super-node i into multiple super-nodes, say $V_i = U_p \sqcup V_{i,1} \sqcup \cdots \sqcup V_{i,d}$, that are connected in a tree topology where the last edge in the path from U_p to each $V_{i,j}$, $j \in [d]$, is set according to the minimum cut between a pivot $p \in U_p$ and a corresponding $u_{i,j} \in V_{i,j}$, where all these cuts are computed in the same auxiliary graph. We call this an *expansion step* and super-node U_p is called the *expansion center*; see Figure 5.2 for illustration. Each iteration of our algorithm applies such an expansion step to every super-node in the intermediate tree $\mathcal{T}$. These iterations can also be viewed as recursion, and thus each expansion step occurs at a certain recursion depth, which will be bounded by our construction.

To prove that our algorithm is correct, we will show that every expansion step corresponds to a valid sequence of Gomory-Hu steps. Just like in the Gomory-Hu algorithm, our algorithm relies on minimum-cut computations in auxiliary graphs, although it will make multiple queries on the same auxiliary graph. This alone does not guarantee overall running time $\tilde{O}(m)$, because in some scenarios the total size of all auxiliary graphs at a single depth is much bigger than m. For example, if $\mathcal{T}^*$ consists of two stars of size $n/3$ connected by a path of length $n/3$, and G is similar but has in addition all possible edges between the stars (with low weight), the total size of all auxiliary graphs would be $\Omega(n^3)$. We overcome this obstacle using a *capacitated auxiliary graph* (CAG), which is the same auxiliary graph as in the Gomory-Hu algorithm, but with parallel edges merged into a single edge with their total capacity. We will show (in Lemma 5.3.12) that the total size of all CAGs at a single depth is linear in m.

Another challenge is to bound the recursion depth by $O(\log n)$. A partition in the Gomory-Hu algorithm might be unbalanced, where in our algorithm, this issue comes into play by a poor choice of a pivot; for example, in a star graph with edge-capacities $1, \ldots, n-1$, if the pivot p is the leaf incident to the edge of capacity 1, then the minimum cut between p and

[10]Even though the perturbation algorithm is Monte Carlo, our algorithm can still be made Las Vegas since if a random perturbation fails Assumption 5.3.3, then our algorithm could encounter two crossing cuts, but it can identify this situation and restart the algorithm with another perturbation.

any other node is the same $(\{p\}, V \setminus \{p\})$, giving little information on how to partition V and make significant progress. Observe however that a random pivot would work much better in this example; more precisely, a set of $O(\log n)$ random pivots contains, with high probability, at least one pivot p for which the minimum cuts between p and each of the other nodes will partition V into super-nodes that are all constant-factor smaller, thus our expansion step will decrease the super-node size by a constant factor. But notice that even if a pivot p is given, we still need to bound the time it takes to partition the super-node. Our algorithm repeatedly computes a minimum cut between p and some other node, such that the time spent on computing this minimum cut is proportional to its progress in reducing $|V_i|$, until $\Omega(|V_i|)$ nodes are separated away from V_i. Altogether, all these minimum cuts (from a single pivot p) take time that is near-linear in the size of the corresponding CAG. It will then follow that the total time of all expansion steps at a single depth is near-linear in the total size of their CAGs, which as mentioned above is linear in m, and finally since the depth is $O(\log n)$, the overall time bound is $\tilde{O}(m)$.

5.3.3 Full Algorithm

To better illustrate our main ideas, we now present our algorithm with a slight technical simplification of employing both Min-Cut and Max-Flow queries. After analyzing its correctness and running time in Section 5.3.4, we will show that Max-Flow queries are not necessary, in Section 5.3.5.

The algorithm initializes $\mathcal{T}$ as a single super-node associated with the entire node set V, and ends when all super-nodes in $\mathcal{T}$ are singletons, supposedly corresponding to the cut-equivalent tree $\mathcal{T}^*$. At every recursion depth in between, the algorithm performs an expansion step in every non-singleton super-node. The expansion of super-node $i \in \mathcal{T}$ of size $n_i = |V_i| \geq 2$, whose CAG is denoted G_i, works as follows. Pick a pivot node $p \in V_i$ uniformly at random, and for every node $u \in V_i \setminus \{p\}$ let $(S_u, V(G_i) \setminus S_u)$ be the minimum *up*-cut in G_i, and let $S'_u = V_i \cap S_u$. In order to compute $|S'_u|$, create in a preprocessing step a copy $\tilde{G}_i$ of G_i, and assuming its edge-capacities are integers (by scaling), connect (in $\tilde{G}_i$) the pivot p to all other nodes $u \in V_i \setminus \{p\}$ by new edges of small capacity $\delta = 1/n^3$. Note that $\tilde{G}$ depends on p but not on u, hence it is preprocessed once per pivot p then used for multiple nodes u. Then for every node $u \in V_i \setminus \{p\}$ compute

$$h_p(u) := [\mathsf{Max\text{-}Flow\text{-}Value}_{\tilde{G}_i}(u,p) - \mathsf{Max\text{-}Flow\text{-}Value}_{G_i}(u,p)]/\delta,$$

which clearly satisfies $h_p(u) = |S'_u|$, and then compute the set

$$V_i^{\leq 1/2}(p) := \{u \in V_i \setminus \{p\} : h_p(u) \leq n_i/2\}.$$

Now repeat picking random pivots until finding a pivot p for which $|V_i^{\leq 1/2}(p)| \geq n_i/4$.

Next, initialize $U_p := V_i$, pick uniformly at random a node $u \in U_p \cap V_i^{\leq 1/2}(p)$, and enumerate the edges in the cut $(S_u, V(G_i) \setminus S_u)$. Partition U_p into two super-nodes, $U_p \cap S_u$ and $U_p \setminus S_u$, connected by an edge of capacity $\mathsf{Max\text{-}Flow\text{-}Value}(u,p)$, then reconnect every edge previously connected to U_p in $\mathcal{T}$ to either $U_p \cap S_u$ or $U_p \setminus S_u$ according to the cut $(V(G_i) \setminus S_u, S_u)$. Repeat the above, i.e., pick another node $u \in U_p \cap V_i^{\leq 1/2}(p)$ and so forth, as long as $|U_p| > 7n_i/8$ (we shall prove that such a node u always exists), calling these nodes $u_1, \ldots, u_d$ in the order they are picked by the algorithm; when $|U_p| \leq 7n_i/8$ is reached, conclude the current expansion step.

Recall that the algorithm performs such an expansion step to every non-singleton super-node (i.e., $n_i \geq 2$) at the current depth, and only then proceeds to the next depth. The base case $n_i = 1$ can be viewed as returning a trivial tree on V_i.

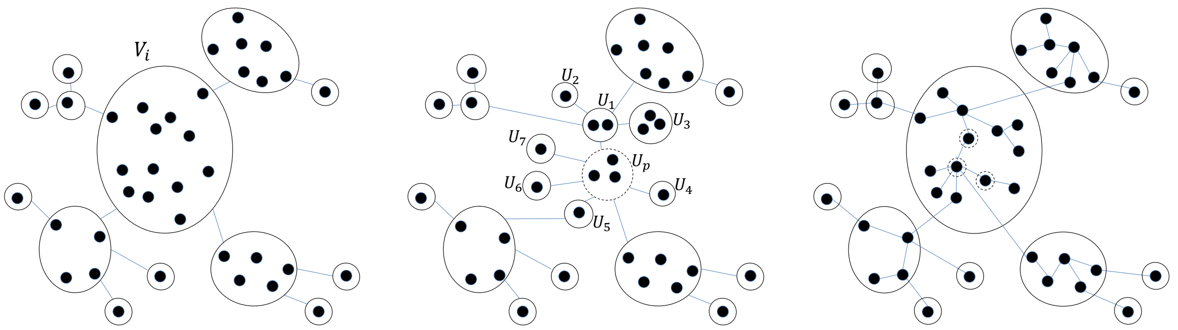

Figure 5.2: The changes to $\mathcal{T}$ by our algorithm. Left: before expansion step of V_i. Middle: after expansion step with expansion center U_p (dashed), and the subtree of $\mathcal{T}$ corresponds to partition $V_i = \bigsqcup_{j=1}^{7} U_j \sqcup U_p$. Right: when the algorithm terminates.

5.3.4 Analysis

We start by showing that whenever our algorithm reports a tree, there exists a Gomory-Hu execution that produces the same tree. Notice that super-nodes at the same depth are disjoint, hence an expansion of one of them does not affect the other super-nodes, and the result of these expansion steps is the same regardless of whether they are executed in parallel or sequentially in any order.

Lemma 5.3.4 (Simulation by Gomory-Hu Steps)**.** *Suppose there is a sequence of Gomory-Hu steps producing tree $\mathcal{T}^{(j)}$, and that an expansion step performed to $V_i \in \mathcal{T}^{(j)}$ produces $\mathcal{T}^{(j+1)}$. Then there is a sequence of Gomory-Hu steps that simulates also this expansion step and produces $\mathcal{T}^{(j+1)}$.*

Proof. Assume there is a truncated execution of the Gomory-Hu algorithm that produces $\mathcal{T}^{(j)}$, we describe next a sequence of Gomory-Hu algorithm's steps starting with $\mathcal{T}^{(j)}$ that produces $\mathcal{T}^{(j+1)}$. Recall that to produce $\mathcal{T}^{(j+1)}$, our algorithm partitions a super-node $V_i \in \mathcal{T}^{(j)}$ into $U_p \sqcup V_{i,1} \sqcup \cdots \sqcup V_{i,d}$, where the last edge in the path from super-node $U_p \in \mathcal{T}^{(j+1)}$ to each super-node $V_{i,k} \in \mathcal{T}^{(j+1)}$ for $k \in [d]$ was set according to the minimum cut between a pivot $p \in U_p$ and a corresponding $u_{i,k} \in U_{i,k}$, at the time of the partition, and these minimum cuts are computed in the same auxiliary graph G_i. Let $u_{i,1}, \ldots, u_{i,d}$ be in the order they are picked by the algorithm, thus if the path between U_p and $u_{i,a}$ in $\mathcal{T}^{(j+1)}$ contains $U_{i,b}$, then $a \leq b$ (We may omit the subscript i when it is clear from the context.)

The Gomory-Hu steps are as follows. Starting with $\mathcal{T}^{(j)}$, for each $k = 1, \ldots, d$, execute a Gomory-Hu step with the pair u_k, p from super-node U_p in $\mathcal{T}$ (we will shortly show that indeed $u_k, p \in U_p$ at that stage), and denote the resulting tree by $\mathcal{T}^{(j),k}$. By convention, $\mathcal{T}^{(j),0} := \mathcal{T}^{(j)}$.

Informally, one may ask why can we carry out multiple Gomory-Hu steps using the same auxiliary graph and circumvent the sequential nature of the Gomory-Hu algorithm? The answer stems from the Gomory-Hu analysis, that for every $s, t \in V_i$ the minimum st-cut in G_i is also a minimum st-cut in G, and from Assumption 5.3.3, which guarantees that the minimum st-cuts in G are unique, and thus do not cross each other. Therefore these cuts may be found all in the same auxiliary graph, and we only need to verify the corresponding Gomory-Hu steps.

More formally, we prove by induction that for every $k \in [0, .., d]$, there is a sequence of Gomory-Hu steps that produces $\mathcal{T}^{(j),k}$. The base case $k = 0$ holds because of our initial assumption that $\mathcal{T}^{(j)}$ can be produced by a sequence of Gomory-Hu steps. For the inductive step, assume that $\mathcal{T}^{(j),k}$ can be produced by a sequence of Gomory-Hu steps. By the analysis of the Gomory-Hu algorithm, for every pair of nodes $s, t \in U_p$ in $\mathcal{T}^{(j),k}$, the minimum st-cut in the auxiliary graph of U_p in $\mathcal{T}^{(j),k}$ is a minimum st-cut in G, and this is correct in particular for the pair our algorithm picks, u_{k+1}, p. By the same reasoning, the minimum $u_{k+1}p$-cut in G_i is also a minimum pu_{k+1}-cut in G. By Assumption 5.3.3, these two cuts are identical, and hence the partition of U_p in $\mathcal{T}^{(j),k}$ that our algorithm performs and the reconnection of the subtrees that it does (based on the minimum $u_{k+1}p$-cut in G_i) is exactly the same as the Gomory-Hu execution would do (based on the minimum $u_{k+1}p$-cut in the auxiliary graph of U_p in $\mathcal{T}^{(j),k}$), resulting in $\mathcal{T}^{(j),k+1}$. Lemma 5.3.4 now follows from the case $k = d$. □

The next corollary follows from Lemma 5.3.4 immediately by induction.

Corollary 5.3.5. *There is a Gomory-Hu execution that outputs the same tree as our algorithm, which by the correctness of the Gomory-Hu algorithm and Assumption 5.3.3, is the cut-equivalent tree $\mathcal{T}^*$.*

We proceed to prove the time bound stated in Theorem 5.3.1. Our strategy is to bound the running time of a single expansion step in proportion to the size of the corresponding CAG, and then bound the total size, as well as the construction time, of all CAGs at a single depth of the recursion. Finally, we will bound the recursion depth by $O(\log n)$, to conclude the overall time bound stated in Theorem 5.3.1.

Lemma 5.3.6. *Assuming $t_p(m) = \tilde{O}(m)$ and $t_{mc}(m) = \tilde{O}(1)$, the (randomized) running time of a single expansion step on V_i, including constructing the children CAGs, and preprocessing it for queries, is near-linear in the size of G_i with probability at least $1 - 1/n^3$.*

Proof. We start with bounding the number of pivot choices. To do that, we use Corollary 5.2.5 with $V_F = V(G_i)$, $V'_F = V_i$, and $H_{G_i}(V_i)$ as the helper graph of G_i on V_i, where the corresponding cuts are the minimum cuts between pairs in V_i. By Corollary 5.2.5, the probability that at least $4 \log n$ random pivots p all satisfy $|V_i^{\leq 1/2}(p)| < n_i/4$, which we call an *unsuccessful* choice of pivot p, is bounded by $1/n^4$. The number of expansion steps is at most $n - 1$, because the final tree $\mathcal{T}$ contains $n - 1$ edges, and each expansion step creates at least one such edge. By a union bound we conclude that with probability at least $1 - 1/n^3$, every expansion step picks a successful pivot within $4 \log n$ trials. Observe that for every choice of p we compute $h_p(u)$ for all $u \in V_i$, which takes time $\tilde{O}(|V_i| + |G_i|)$ for all pivots. We can thus focus henceforth on the execution with a successful pivot p.

We now turn to bound the total time spent on queries in G_i. Let $\mathcal{T}_i^*$ be the subgraph of $\mathcal{T}^*$ induced on V_i. Observe that $\mathcal{T}_i^*$ must be connected, because V_i is a super-node in an intermediate tree of the Gomory-Hu algorithm (see Lemma 5.3.4). Define a function

$\ell : V(\mathcal{T}_i^*) \setminus \{p\} \to E(\mathcal{T}_i^*)$, where $\ell(u)$ is the lightest edge in the path between u and p in $\mathcal{T}_i^*$, and $\ell(p) = \emptyset$ (see Figure 5.3 for illustration); it is well-defined because Assumption 5.3.3 guarantees there are no ties. For an edge $e \in \mathcal{T}_i^*$, we say that e is *hit* if the targets $u_{i,1}, \dots, u_{i,d}$ picked by the expansion step include a node u such that $\ell(u) = e$. Let H_e be an indicator for the event that edge e is hit. In order to bound the total number of nodes and edges in the CAG that participate in minimum-cut queries performed by the expansion step, we first bound the number of edges that are hit along any single path.

Claim 5.3.7. *With high probability, for every path P between a leaf and p in $\mathcal{T}_i^*$, the number of edges in P that are hit is $\sum_{e \in P} H_e \le O(\log n)$.*

Proof. Let $\mathcal{T}_{i,\ell}^*$ be the graph constructed from $\mathcal{T}_i^*$ by merging nodes whose image under ℓ is the same. Observe that nodes that are merged together, namely, $\ell^{-1}(e)$ for $e \in E(\mathcal{T}_i^*)$, are connected in $\mathcal{T}_i^*$, and therefore the resulting $\mathcal{T}_{i,\ell}^*$ is a tree. See Figure 5.3 for illustration. We shall refer to nodes of $\mathcal{T}_{i,\ell}^*$ as *vertices* to distinguish them from nodes in the other graphs. For example, p is not merged with any other node, and thus forms its own vertex.

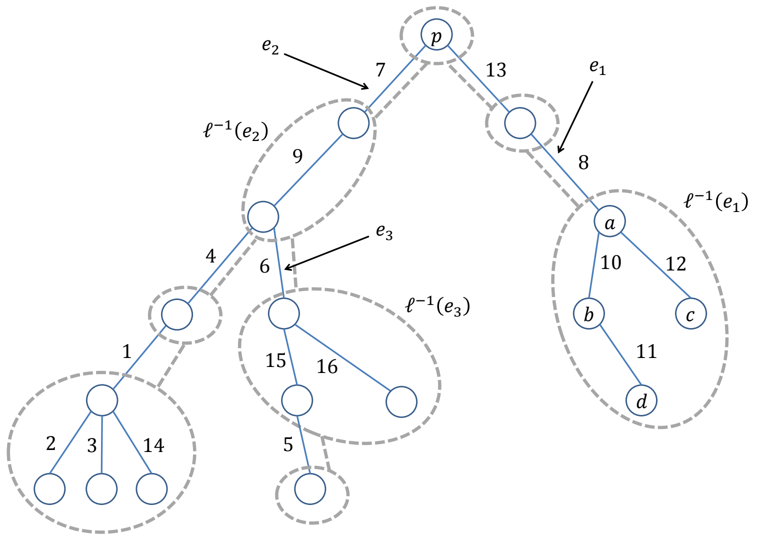

Figure 5.3: An illustration showing $\mathcal{T}_i^*$ with solid blue lines, while the corresponding graph $\mathcal{T}_{i,\ell}^*$ with dashed gray lines. For example, $e_1 = \ell(a) = \ell(b) = \ell(c) = \ell(d)$. The nodes in $\ell^{-1}(e_2)$ are not in $V_i^{\le 1/2}(p)$, and so the expansion step never picks any of them as a sink. After picking any node from $\ell^{-1}(e_3)$, a new super-node containing $\ell(e_3)$ (and possibly the vertex below as well) is formed.

For sake of analysis, fix a leaf in $\mathcal{T}_{i,\ell}^*$, which determines a path to the root p, denoted P_ℓ, and let us now bound the number of nodes picked (by the expansion step) from vertices in P_ℓ.

Claim 5.3.8. *With high probability, the total number of nodes u picked by the algorithm from vertices in P_ℓ is at most $O(\log n)$.*

Proof. We will need the following two observations regarding $\mathcal{T}^*_{i,\ell}$.

Observation 5.3.9. *No vertex in $\mathcal{T}^*_{i,\ell}$ contains nodes from both $V_i^{\leq 1/2}(p)$ and $V_i \setminus V_i^{\leq 1/2}(p)$.*

This is true because all nodes u in the same vertex $\ell^{-1}(e)$ have the same minimum *up*-cut in G, which is a basic property of the cut-equivalent tree $\mathcal{T}^*$, and thus all these nodes will have the same S_u and the same S'_u computed in the CAG G_i.

Observation 5.3.10. *The vertices that contain nodes in $V_i^{\leq 1/2}(p)$ form a prefix of the path P_ℓ.*

This is true by monotonicity of $|S_x|$ as a function of the hop-distance of x from p in P_ℓ, denoted P'_ℓ.

The algorithm only picks nodes from $V_i^{\leq 1/2}(p)$, thus it suffices to bound the nodes picked from (the vertices along) the prefix P'_ℓ. Fix a list π of the nodes in (vertices in) P'_ℓ in increasing order of their hop-distance from p in P_ℓ, Now recall that the targets $u_{i,1}, \ldots, u_{i,d}$ are chosen sequentially, each time uniformly at random from $U_p \cap V_i^{\leq 1/2}(p)$ for the current U_p. Initially, U_p contains all the nodes in π (but may contain also nodes outside the path P_ℓ). Now each time a target u is chosen, some nodes are separated away from U_p. Define the list π' to be the restriction of π to nodes currently in U_p; notice that U_p and π' change during the random target choices, but π is fixed. We can classify the randomly chosen target u into three types.

1. u is not from the current list π': In this case π' does not change. We call this a "don't care" event, because we shall ignore this choice.
2. u is from the current list π': In this case π' is shortened into a prefix of π' that *does not* contain u. We now have two subcases:

   2.a. u is from the first half of π': Then π' is shortened by factor at least 2. We call this event "big progress".

   2.b. u is from the second half of π': We call this event "small progress".

Now to complete the proof of Claim 5.3.8, consider the random process of choosing the targets u. To count the number of targets u from P_ℓ, we can ignore targets of type 1 and focus on targets of type 2, in which case type 2a occurs with probability at least 1/2. As the initial list π has length at most n, with high probability the random process terminates within $16 \log n$ steps (counting only targets of type 2). [11] □

Proceeding with the proof of Claim 5.3.7, suppose the path P consists of nodes $v_1, \ldots, v_k = p$ where v_1 is the leaf. Then the path P_ℓ consists of $\ell^{-1}(\ell(v_1)), \ldots, \ell^{-1}(\ell(v_k))$ restricted to distinct vertices. Note that whenever an edge e in P that is hit, some target u is picked from $\ell^{-1}(e)$ and in particular from P_ℓ. By Claim 5.3.8, with high probability the number of target nodes picked from P_ℓ is bounded by $O(\log n)$, implying that also the number of hit edges in P is bounded by $O(\log n)$. Finally, Claim 5.3.7 follows by applying a union bound over all (at most n) leaves. □

Next, we use Claim 5.3.7 to bound the total running time of an expansion step.

[11] The similar but different idea that the minimum cuts from a uniformly random node p partition the auxiliary graph in a balanced way with high probability, which allows bounding the recursion depth by analyzing the maximal length of paths in the recursion tree, appears in Lemma 35 and Theorem 11 in [BCH$^+$08].

Claim 5.3.11. *An internal iteration in the expansion step, that partitions a super-node U_p into $U_p \setminus S_u$ and $U_p \cap S_u$, takes time $\tilde{O}(|S_u| + k^i_{up})$, where k^i_{up} is the number of edges in the minimum up-cut $(V(G_i) \setminus S_u, S_u)$.*

Proof. Using the Min-Cut data structure, the algorithm spends $\tilde{O}(k^i_{up})$ time for finding the edges in the minimum *up*-cut $(S_u, V(G_i) \setminus S_u)$, where we denote their number by k^i_{up}. When partitioning a super-node U_p, the algorithm does not explicitly list the nodes in $U_p \setminus S_u$ as this would take too much time. Instead, it only lists the nodes in $U_p \cap S_u$, i.e., those that are separated from U_p, as follows. We first find S_u by using Claim 5.2.9 on G_i, with terminals initialized to $V_T := V(G_i)$, and queries to $S := S_u$. Observe that in our case S_u is connected (i.e., $S(u) = S_u$) as otherwise there would have been a subset $\tilde{S}_u \subset S_u$ such that $c(S'_u) < c(S_u)$, contradicting the minimality of $c(S_u)$.

Second, we enumerate the nodes in S_u and test for membership in U_p, to find $U_p \cap S_u$. Recall that updating the intermediate tree $\mathcal{T}$ requires reconnecting each edge that was initially incident to super-node U_p, to one of the two new super-nodes $U_p \setminus S_u$ and $U_p \cap S_u$. Thus, we discuss this reconnection process next.

Throughout the expansion step, we maintain a list L of all super-nodes that are adjacent to U_p, starting with the super-nodes $G_i \setminus V_i$. Technically, for each super-node V_j adjacent to U_p it is stored by a representative node from V_j and a pointer to V_j. In order to reconnect subtrees after partitioning $U_p \cap S_u$ out of U_p, the algorithm finds which super-nodes in L are in S_u. This is done by enumerating the nodes in $S_u \cap V(G_i)$ and testing for membership in L. Then, connect those super-nodes to the new super-node $U_p \cap S_u$ in $\mathcal{T}$, and finally update L to reflect the reconnection. At the end, $U_p \setminus S_u$ is connected to the remaining subtrees. This proves Claim 5.3.11. □

We continue with the proof of Lemma 5.3.6, that the total time for an expansion step is bounded. We may assume henceforth that the $O(\log n)$ bound in Claim 5.3.7 holds, as it occurs with high probability. The number of times a node $u \in V(G_i)$ is queried (when it belongs to some S_v) is equal to the number of hit edges in its path to the pivot p in $\mathcal{T}^*_i$, which we just assumed to be bounded by $O(\log n)$. The number of times an edge $e \in E(G_i)$ is queried is equal to the number of hit edges in $\mathcal{T}^*_i$ along the two paths from e's ends to the pivot p, which we just assumed to be bounded by $O(\log n)$. Altogether, the time it takes to scan the cuts $S_{u_{i,1}}, \ldots, S_{u_{i,d}}$ and the corresponding super-nodes $V_{i,1}, \ldots, V_{i,d}$ that are separated away from V_i is bounded, by Claim 5.3.11, by

$$\tilde{O}\Big(\sum_{j=1}^{d} |S_{u_{i,j}}| + k^i_{u_{i,j}p}\Big) \leq \tilde{O}\Big(|V(G_i)| + |E(G_i)|\Big).$$

Finally, observe that the total time it takes to construct the CAGs of any super-node V_i's children in a single expansion step is linear in the size of V_i's CAG. This completes the proof of Lemma 5.3.6. □

Next, we show that the total size of all CAGs at a certain depth is bounded by $O(m)$. In fact, we show it for partition trees, which generalize the intermediate trees produced by our algorithm. A *partition tree* T of a graph $G = (V, E)$ is a tree whose nodes $V_1, \ldots, V_l$ are super-nodes of G and form a partition $V = V_1 \sqcup \cdots \sqcup V_l$. Clearly, our intermediate tree $\mathcal{T}$ is a partition tree, and so we are left with proving the following lemma.

Lemma 5.3.12. *Let $G = (V, E)$ be an input graph, and let T be a partition tree on super-nodes $V_1, \ldots, V_l$. Then the total size of the corresponding CAGs $G_1, \ldots, G_l$ is at most $2n + 3m = O(m)$.*

Proof. Root T at an arbitrary node r and direct all edges away from r. Now charge each edge e in a CAG G_i to some graph edge $uv \in E(G)$ that contributes to its capacity, picking one arbitrarily if there are multiple such edges. Let P_{uv} be the path in T between the two super-nodes V_u and V_v that contain u and v, respectively, and observe that super-node V_i must lie on this path, see Figure 5.4 for illustration.

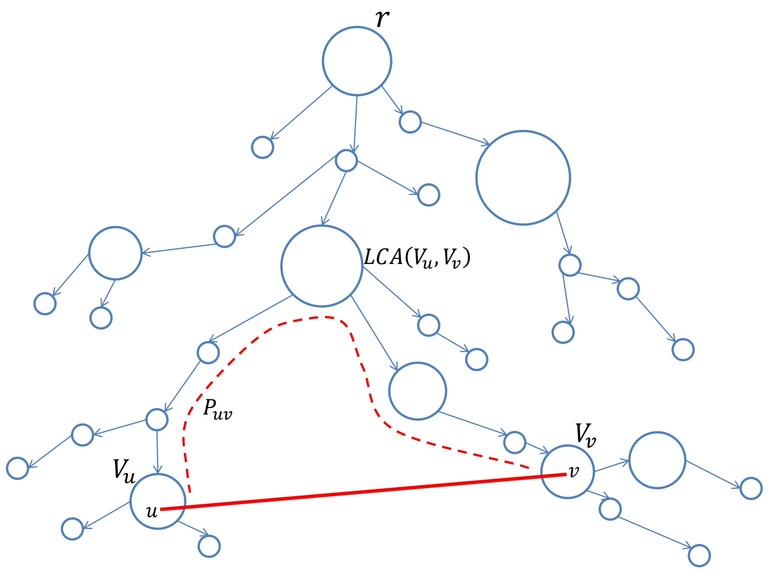

Figure 5.4: An illustration of the partition tree T rooted at r. The thick red line depicts a graph edge $uv \in E(G)$ that is being charged. The dashed red curve depicts P_{uv}, the path in T between super-nodes V_u and V_v.

To bound the total charge for a single graph edge $uv \in E(G)$, observe that it cannot be charged by two edges e', e'' in the same CAG G_i, it thus suffices to count how many different CAGs contribute to the charge of uv. We split this into three cases.

1. V_i is an endpoint of P_{uv} (i.e., $V_i = V_u$ or $V_i = V_v$): An edge $uv \in E(G)$ can be charged in this manner at most twice (over all CAGs), namely, by one edge in G_u and one in G_v. Thus, the total charge over all $uv \in E(G)$ is at most $2m$.

2. V_i is the least common ancestor, abbreviated LCA, of V_u and V_v in T: An edge $uv \in E(G)$ can be charged in this manner at most once (over all CAGs). Thus, the total charge over all $uv \in E(G)$ is at most m.

3. V_i is not an endpoint of P_{uv} nor it is the LCA of V_u and V_v: In this case, exactly one of V_u and V_v is a descendant of V_i. We bound the number of such edges e (over all CAGs) directly, i.e., without charging to uv, as follows.

Let d_i be the degree of V_i in the tree T. Recall that the CAG G_i is obtained from G by merging the nodes in $V \setminus V_i$ into exactly d_i nodes, one for each neighbor of V_i in T, and one of these d_i nodes in G_i, denote it $\hat{x}_i$, is the merger of all the nodes from all the super-nodes V_j that are non-descendants of V_i. (see section 5.3.2). It follows that an edge e in G_i (in this case) connects this $\hat{x}_i$ to one of the other $d_i - 1$ nodes mentioned above, and clearly there are at most $d_i - 1$ such edges. By summing over all the CAGs $G_1, \ldots, G_l$, the total number of such edges e is at most $\sum_i (d_i - 1) \leq 2n$.

Altogether, the total size of all the CAGs is at most $2m + m + 2n = O(m)$, as claimed. □

We are now ready to prove the main Theorem.

Proof of Theorem 5.3.1 under the assumption on Max-Flow queries. To simplify matters, let us assume henceforth that $t_p(m) = \tilde{O}(m)$ and $t_{mc}(m) = \tilde{O}(1)$. The general case is analyzed similarly and results in the time bound $\tilde{O}(t_p(m) + m \cdot t_{mc}(m))$ stated in Theorem 5.3.1 for the following reasons. The preprocessing time is performed $\tilde{O}(1)$ times per CAG, hence the total preprocessing time over all CAGs that the algorithm constructs is at most $\tilde{O}(t_p(m))$, the first summand above. The total size of all answers to all queries at a single depth is near-linear in the total size of all CAGs at this depth; hence over all depths it is bounded by $\tilde{O}(m \cdot t_{mc}(m))$, the second summand above.

First, assume the perturbation attempt from Section 5.3.6 is successful. By Lemma 5.3.6 the total time spent at each super-node V_i is near-linear in the size of G_i, and thus by Lemma 5.3.12, the total time spent at each recursion depth is bounded by $O(m)$. By the definition of the algorithm, at each super-node V_i during the recursion, $\Theta(|V_i|)$ nodes are partitioned away from V_i, and so by Lemma 5.3.12, $\Theta(n)$ nodes are partitioned away from all CAGs at this depth, thus after the $O(\log n)$ depth, each super-node V_i is a singleton, concluding Theorem 5.3.1 in this case.

Second, if the perturbation attempt from Section 5.3.6 is unsuccessful, which happens with probability at most $1/n^3$, and two cuts are crossing each other, then we would identify that and restart the algorithm. By Lemma 5.3.6, with probability at most $1/n^3$ the number of incorrect pivots exceeds $O(\log n)$, and by a union bound with the probability of a failed perturbation attempt, the running time of the algorithm is bounded by $\tilde{O}(m)$ with high probability. □

5.3.5 Lifting the Assumption on Max-Flow Queries

Recall that our goal is to construct a cut-equivalent tree using access to Min-Cut queries. So far we have assumed that we also have access to Max-Flow queries. In this subsection we show how to lift this additional assumption. We will change the algorithm and the analysis slightly, as follows.

First, at each expansion step, run the algorithm on $4 \log n$ preprocessed copies of G_i, each on one of the randomly picked pivots. Similar to our calculation from the original proof, with high probability, for every expansion step throughout the execution, at least one of the corresponding graphs will have a successful pivot. We will make sure that an unsuccessful pivot will never output a wrong tree; it may only keep running indefinitely (until we halt it). Since with high probability at least one of the graphs is of a successful pivot, this only incurs a factor of $\tilde{O}(1)$ to the running time.

Second, instead of picking a node $u \in U_p \cap V_i^{\leq 1/2}(p)$ at random as in the original algorithm, pick $4\log_{8/7} n$ nodes from U_p and use Claim 5.2.9 on $4\log_{8/7} n$ copies of G_i, simultaneously, each for one of the chosen nodes u, to test if $|S'_u| \leq n_i/2$. If all nodes were unsuccessful choices, draw another set of $4\log_{8/7} n$ nodes. Continue to draw batches until at least one node is successful. Then, for an arbitrary successful node u, use Claim 5.2.9 to find the k^i_{up} edges in the minimum *up*-cut, and the nodes in S'_u.

Since the probability for a single node u chosen at random to satisfy $|S'_u| \leq n_i/2$ is always at least $1/8$, and as we pick $4\log_{8/7} n$ nodes uniformly at random each time, we get that: with probability at least $1 - (7/8)^{4\log_{8/7} n} = 1 - 1/n^4$, at least one of the $4\log_{8/7} n$ chosen nodes is successful. By a union bound over the maximal number of partitions in expansion steps throughout the execution, i.e. internal iterations of expansion steps (at most n), we get that with probability at least $1 - 1/n^3$ each one of the batches results in at least one of the $4\log_{8/7} n$ nodes in the batch is successful. Hence, the only part of the proof that needs to be further addressed is Claim 5.3.7. In particular, we prove the following variant of the claim.

Claim 5.3.13. *With high probability, for every path P between a leaf and p in $\mathcal{T}_i^*$, the total number of edges in P that are hit is at most $O(\log^2 n)$.*

Proof. We mention the differences from the proof of the original Claim 5.3.7. The classification of the choice of a random target u into three types is as follows.

1. (Similar to before) u is not from the current list π': In this case π' does not change. We call this a "don't care" event, because we shall ignore this choice.
2. u is from the current list π': In this case π' is shortened into a prefix of π' that *does not* contain u. We now have two subcases:

   2.a. u is from the first $1 - 1/(3\log_{8/7} n)$ fraction of π': Then π' is shortened by factor at least $1/(3\log_{8/7} n)$. We call this event "big progress".

   2.b. u is from the complement part of π': We call this event "small progress".

Here, we have a random process in which type 2a occurs with probability at least $1 - 1/(3\log_{8/7} n)$, and therefore with high probability it terminates within $64\log_{8/7} n \ln n$ steps (these steps count only targets of type 2). We conclude that with high probability, every such path has at most $64\log_{8/7} n \ln n = O(\log^2 n)$ nodes chosen from its vertices. □

We proceed to the proof of Theorem 5.3.1, highlighting the differences.

Proof of Theorem 5.3.1. With high probability, at each expansion step at most $O(\log n)$ unsuccessful pivots are chosen before picking a successful one. At each level, we spend at most $t_p(m)$ time for the preprocessing of the min-cut data structures for fixed sources, and so unsuccessful pivots only incur a factor $\tilde{O}(1)$ on the running time. Thus, the proof of Theorem 5.3.1 is concluded. □

5.3.6 Unique Cut-Equivalent Tree via Pertubation

The following proposition shows that by adding small capacities to the edges, we can assume that G has one cut-equivalent tree $\mathcal{T}^*$ (see also [BENW16, Preliminaries]).

Proposition 5.3.14. *One can add random polynomially-bounded values to the edge-capacities in G, such that with high probability, the resulting graph G' has a single cut-equivalent tree $\mathcal{T}^*$ with $n-1$ distinct edge weights, and moreover the same $\mathcal{T}^*$ (with edge weights rounded back) is a valid cut-equivalent tree also for G.*

Proof. We use the following well known lemma.

Lemma 5.3.15 (The Isolation Lemma [MVV87])**.** *Let h and H be positive integers, and let $\mathcal{F}$ be an arbitrary family of subsets of the universe $[h]$. Suppose each element $x \in [h]$ in the universe receives an integer weight $w(x)$, each of which is chosen independently and uniformly at random from $[H]$. The weight of a set S in $\mathcal{F}$ is defined as $w(S) := \sum_{x \in S} w(x)$. Then, there is probability at most h/H that more than one set in $\mathcal{F}$ will attain the minimum weight among them.*

Consider $s, t \in V$. Using the lemma above with $\mathcal{F}$ the set of all minimum st-cuts in G, $h := m$, and $H := n^7$, we would get that there is probability at most $1/n^5$ that more than one cut separating s and t will attain the minimum capacity among them (i.e. will be a minimum st-cut). However this might drastically change the capacity of the edges (and cuts), so we divide all added weights by n^{10}. In other words, we add a number from $\{1/n^{10}, \ldots, n^7/n^{10}\}$ uniformly at random to the capacity of every edge in G to get that with probability at most $1/n^5$, the pair s, t have more than one minimum st-cut, and also the capacity of the cut remains close to its original value. By a union bound over all pairs in V there is a probability of at most $1/n^3$ for at least one pair to have more than one minimum cut. Next, the probability for two minimum-cuts $(S_u, V \setminus S_u)$ and $(S_w, V \setminus S_w)$ separating two different pairs of nodes u, u' and w, w', respectively, to have the same value after the perturbation is small. Without loss of generality, let e be an edge in the cut $(S_u, V \setminus S_u)$ but not in $(S_w, V \setminus S_w)$. Conditioning on the values of all other edges, e could have at most one value that makes the cuts' values equal. Since each value is drawn with probability $1/n^7$, by a union bound on all pairs of pairs of node in V, the probability that two different pairs of nodes that have different minimum cuts but had the same value in G will have also the same value in G' (i.e., after the perturbation) is at most $1/n^3$. Finally, by applying a union bound again, with probability at least $1 - 1/n^2$ none of the events happen, that is every pair has a unique minimum cut, and no two pairs of nodes have two different minimum-cuts with the same value.

Since the value of every cut in G' is bigger by at most $m \cdot 1/n^3 \leq 1/n$ than its original value, and assuming the edge-capacities in G are integers (by scaling), the minimum st-cut in G' is smaller than any non-minimum st-cut, that is a cut separating s and t that is not the minimum one in G, and also the value of any non-minimum st-cut in G' is bigger by at least $1 - 1/n$ than the minimum st cut in G. Hence, T^* is a valid cut-equivalent tree for G, and by removing the added weights from T^* we have also the original cut values. This completes the proof of Proposition 5.3.14. □

5.4 Algorithm for an Output Sensitive Data Structure

For completeness, we show here that designing an output sensitive data structure for minimum-cuts can be reduced to the construction of cut-equivalent trees, i.e. the opposite direction than in Section 5.3.

Theorem 5.4.1. *Given a capacitated graph $G = (V, E, c)$ on n nodes, m edges, and a cut-equivalent tree T of G, there is a deterministic data structure that after preprocessing in time $\tilde{O}(m)$, can report for a query pair $s, t \in V$, the edges in a minimum st-cut in time $\tilde{O}(output)$.*

We first give an overview of the reduction. Consider a tour $t_1, \ldots, t_{2n-1} = t_1$ on (the nodes of) the tree T, starting at an arbitrary node t_1 and following a DFS (i.e., going "around" the tree and traversing each edge twice). Now assign each graph edge $e = (w, w') \in E$ two points p^1, p^2 in a two-dimensional grid of size $(2n-1) \times (2n-1)$, as follows. One point p^1 has x and y coordinates according to the first time the tour visits w and w', respectively; the other point p^2 has the same coordinates but in the opposite order. See Figure 5.5 for illustration.

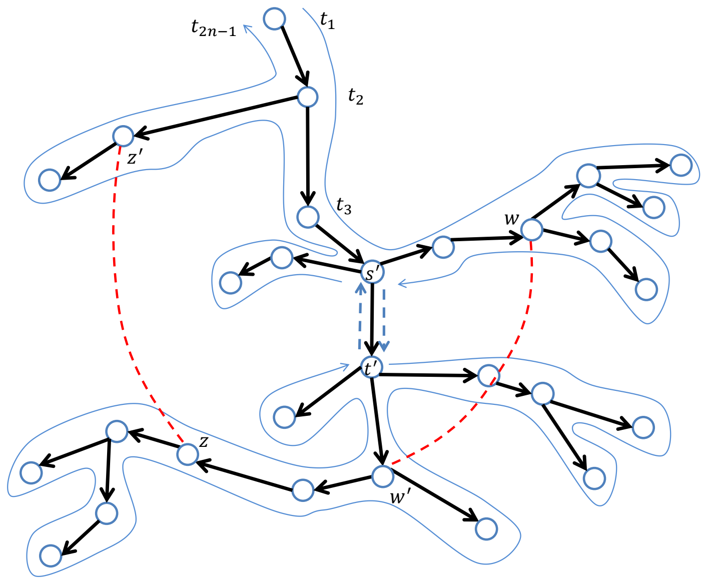

Figure 5.5: An illustration of the tour on T and how the edges E are mapped to grid points P. The rooted tree T is depicted by black arrows, and the tour by a solid blue line except for one tree edge (s', t') that is dashed. We also show two edges of the graph G that have exactly one endpoint in the subtree under t', depicted by dashed red lines. They are mapped to grid points $p^2(w, w')$ inside rectangle (5.1), and $p^1(z, z')$ inside rectangle (5.2).

Given a query pair $s, t \in V$, the algorithm first finds the lightest edge $(s', t') \in E(T)$ in the unique st-path in T. It then reports all the graph edges in the cut corresponding to removing (s', t') from T, using the following observation. View T as rooted at t_1 (where the tour begins), and assume without loss of generality that $s' = \text{parent}(t')$. Then the subtree under t' is visited exactly in the interval $I_{s't'} := [\text{FirstVisit}(t'), \text{LastVisit}(t')]$ where for a node $q \in V$,

$$\text{FirstVisit}(q) := \min\{k \in [2n-1] : \ t_k = q\},$$
$$\text{LastVisit}(q) := \max\{k \in [2n-1] : \ t_k = q\}.$$

As a result, every graph edge e that crosses the cut corresponding to (s', t') has exactly one endpoint inside the interval $I_{s't'}$ (more precisely, all its visits are inside that interval) and one endpoint outside that interval (actually, all its visits are outside). Finally, we define two rectangles in the grid that contain exactly the points corresponding to edges of this cut, and employ a known algorithm to report all the points (edges of G) inside these rectangles.

Proof of Theorem 5.4.1. The preprocessing algorithm works as follows. Given G and its cut-equivalent tree T, construct a tour $t_1, \dots, t_{2n-1}$ on T as described in the overview. Then, for every graph edge $(w, w') \in E$, create two points

$$p^1(w, w') := (\text{FirstVisit}(w), \text{FirstVisit}(w')),$$
$$p^2(w, w') := (\text{FirstVisit}(w'), \text{FirstVisit}(w)).$$

Store the set P of the $2m$ points created in this manner in a data structure that supports range queries (as explained below).

Given a pair of nodes $s, t \in V$ as a query for minimum st-cut, the algorithm first finds the lightest edge in the unique st-path between in T in $\tilde{O}(1)$ time, denoted (s', t') where we assume without loss of generality that $s' = \text{parent}(t')$ (recall we view t_1 as the root of T). The algorithm then reports all the points in P that lie inside the two rectangles

$$[\text{FirstVisit}(t'), \text{LastVisit}(t')] \times [1, \text{FirstVisit}(t') - 1], \tag{5.1}$$
$$[\text{FirstVisit}(t'), \text{LastVisit}(t')] \times [\text{LastVisit}(t') + 1, 2n - 1]. \tag{5.2}$$

To see why this output is correct, observe that these two rectangles are disjoint, and that their union is exactly $I_{s't'} \times \overline{I_{s't'}}$ (using the notation from the overview). Thus, points of P inside their union correspond precisely to edges in E with exactly one endpoint visited in the interval $I_{s't'}$, i.e., exactly one endpoint in the subtree under t'. Moreover, an edge e can be reported at most once, because it cannot be that both $p^1, p^2 \in I_{s't'} \times \overline{I_{s't'}}$.

Reporting all the points inside these two rectangles could be done by textbook approach through range trees in time $O(k + \log n)$ [PS85], where k is the output size which for us is the number of edges in the cut. The preprocessing time of [PS85] for p points is $O(p \log p)$, and so the preprocessing time of our data structure is $O(m \log m)$, and the query time is $\tilde{O}(\mathit{output})$, where *output* is the number of edges in the output cut.

□

5.5 Algorithm for Flow-Equivalent Trees

In this section we prove that $O(n \log n)$ queries to a Max-Flow-Value oracle are enough to construct a flow-equivalent tree with high probability. This is analogous to the Gomory-Hu algorithm, which constructs a cut-equivalent tree using minimum-cut queries. Let $\mathcal{F}$ be a graph family that is closed under perturbation of edge-capacities, and suppose that for every graph in $\mathcal{F}$ with m edges, after $t_p(m)$ preprocessing time, Max-Flow-Value queries could be answered in time $t_{mf}(m)$. The following is the main result of this section, which is a consequence of Theorem 5.5.3 below. We use the term Min-Cut data structure as in Section 5.3, although we only need here queries for the value (not an actual cut).

Theorem 5.5.1. *Given a capacitated graph $G = (V, E) \in \mathcal{F}$ with n nodes and m edges, as well as access to a deterministic Min-Cut data structure for $\mathcal{F}$ with running times $t_p(m), t_{mf}(m)$, one can construct a flow-equivalent tree for G in time $O(t_p(m) + n \log n \cdot t_{mf}(m) + n \log^2 n)$ with high probability.*

Similar to Section 5.2.5, Theorem 5.5.1 could be adjusted to handle randomized Min-Cut data structures as well.

One application of the above theorem is to graphs with treewidth bounded by (a parameter) t, for which Arikati, Chaudhuri, and Zaroliagis [ACZ98] obtain $t_p = n \log n \cdot 2^{2^{O(t)}}$ and $t_{mf} = 2^{2^{O(t)}}$, and thus our algorithm constructs a flow-equivalent tree on such graphs in time $\tilde{O}_t(n)$, which was not known before.

Corollary 5.5.2. *There is a randomized algorithm that given a capacitated graph G with n nodes and treewidth at most t, constructs with high probability a flow-equivalent tree for G in time $O(n \log n \cdot 2^{2^{O(t)}})$.*

Our main tool can be described as a theorem about recovering ultrametrics. This is stated formally in Theorem 5.5.3, whose proof appears in Section 5.5.1. But we first recall some standard terminology (see also [GV12]). Let (V, dist) be a finite metric space. (which means that distances are non-negative, symmetric, satisfy the triangle inequality, and are zero between, and only between, every point and itself). It is called an *ultrametric space* if in addition

$$\forall u, v, w \in V, \qquad \mathrm{dist}(u, w) \leq \max\{\mathrm{dist}(u, v), \mathrm{dist}(v, w)\}. \tag{5.3}$$

It is easy to see that (5.3) is equivalent to saying that the two largest distances in every "triangle" u, v, w are equal.

A *representing tree* for an ultrametric $(V, dist)$ is a rooted tree $T = (V_T, E_T)$ in which the set of leaves $L \subseteq V_T$ is (a copy of) V, and every internal node (non-leaf) $z \in V_T \setminus L$ has a label $\mathrm{label}_T(z) \in \mathbb{R}^+$. Moreover, the labels along every root-to-leaf path are monotonically decreasing. For two leaves $u, v \in L$, let $T(u, v)$ denote the label of their LCA in T. It is easy to see that $\mathrm{dist}(u, v) = \mathrm{label}_T(u, v)$ is an ultrametric on L, and in particular satisfies (5.3). Without loss of generality, we further assume throughout that that every internal node $v \in V_T \setminus L$ has at least two children.

Theorem 5.5.3. *There is a randomized algorithm that, given oracle access to distances in an ultrametric on a set of n points where the $\binom{n}{2}$ distances have exactly $n-1$ distinct labels, constructs a representing tree of the ultrametric, and with high probability it runs in time $O(n \log n \cdot Q(n) + n \log^2 n)$ using $O(n \log n)$ distance queries, where $Q(n)$ is the time to answer a query.*

Proof of Theorem 5.5.1. Given a graph $G = (V, E)$, we use Proposition 5.3.14 (proved in Section 5.3) to perturb the edge-capacities, and thus we assume henceforth that G has a single cut-equivalent tree with $n-1$ distinct capacities on its edges. Let $N = (V, E')$ be a complete graph, where the weight of every edge (u, v) is $\mathsf{Max\text{-}Flow\text{-}Value}(u, v)$. It is well-known that N with each edge weight inverted, denoted N', is an ultrametric (see [GH61] or Proposition 5 in [GV12]). Since the cut-equivalent tree of G has $n-1$ distinct capacities on its edges, it must be that for the constructed ultrametric, the $\binom{n}{2}$ distances have exactly $n-1$ labels, and so we can apply Theorem 5.5.3 to recover a representing tree $T_{N'}$ of N' in total time $O(t_p(m) + n \log n \cdot t_{mf}(m) + n \log^2 n)$ with high probability of success.

Finally, construct a path P that is a flow-equivalent tree for G, by the following recursive process, resembling a post-order traversal of the tree $T_{N'}$. Given a node r of $T_{N'}$ (initially r is the root), let u, v be its two children, and let T_u, T_v be the subtrees rooted at u, v, respectively. By applying this procedure recursively on u, compute a path P_u that is a flow-equivalent tree for the leaves of T_u, and similarly compute a path P_v for T_v. Now chose arbitrarily one endpoint of P_u and one endpoint of P_v, and connect them by an edge whose capacity is the label of r in $T_{N'}$, and return the resulting path P.

The proof that this process computes a flow-equivalent tree of $T_{N'}$ follows easily by induction. The main observation is that for every two leaves $a \in T_u, b \in T_v$, their LCA in $T_{N'}$ is r and thus $\mathsf{Max\text{-}Flow\text{-}Value}(u, v)$ is the smallest among all pairs of leaves under r, and it follows by induction that the new edge connecting P_u and P_v will have minimum weight among all the edges between a and b in P. The time to construct the path is linear in the size of $T_{N'}$, and this concludes the proof of Theorem 5.5.1. □

5.5.1 Recovering Ultrametrics

Proof of Theorem 5.5.3. Denote the input ultrametric by (V, dist). The algorithm works recursively as follows, starting with $V' = V$. Given a subset $V' \subseteq V$ of size $n' \geq 2$ of points in an ultrametric, pick a pivot point $p \in V'$ uniformly at random, query the distance from p to all other points in V', and enumerate V' as $p = q_1, q_2, \dots, q_{n'}$ in order of non-decreasing distance from p. Repeat picking pivots until finding a pivot p for which

$$\mathrm{dist}(q_{\lceil n'/4 \rceil}, p) < \mathrm{dist}(q_{\lceil n'/2 \rceil + 1}, p). \tag{5.4}$$

We assumed $n' \geq 2$, as in the base case $n' = 1$ the algorithm returns a trivial tree on V'. Next, find $s \in [\lceil n'/4 \rceil, \lceil n'/2 \rceil]$ such that $\mathrm{dist}(q_s, p) < \mathrm{dist}(q_{s+1}, p)$, partition V' into $V'_{\leq s} = \{q_1, \dots, q_s\}$ and $V'_{>s} = \{q_{s+1}, \dots, q_{n'}\}$ (see Figure 5.6). Now recursively construct trees $T'_{\leq s}$ and $T'_{>s}$ representing the ultrametrics induced on $V'_{\leq s}$ and $V'_{>s}$. By Claim 5.5.4 below, each of the constructed trees $T'_{\leq s}$ and $T'_{>s}$ is binary, and its internal nodes have distinct labels.

Finally, connect the tree $T'_{\leq s}$ "into" $T'_{>s}$ as follows. Scan in $T'_{>s}$ the path from the leaf q_{s+1} to the root, and create a new node u_{s+1} with label $\mathrm{dist}(q_{s+1}, p)$ immediately after the last node with a smaller label on this path (by subdividing an existing edge, or adding a parent to the root to form a new root). Then connect $T'_{\leq s}$ under this new node u_{s+1}, and return the combined tree, denoted $T'_{V'}$, as the output.

Claim 5.5.4. *For every $V' \subseteq V$, every representing tree $T_{V'}$ of the ultrametric (V', dist) is binary, and each of its internal nodes has a distinct distance label.*

Proof. We first claim that the number of distinct distances in the ultrametric induced on V' is at least $n' - 1$. Indeed, consider starting with the entire ultrametric (V, dist), which has exactly $n-1$ distances, and removing the points in $V \setminus V'$ one by one. Each removed point can decrease the number of distinct distances by at most 1, because if removing point z eliminates two distinct distances, say to points x_1 and x_2, then the "triangle" z, x_1, x_2 has three distinct distances, in contradiction with (5.3). Since (V, dist) has exactly $n - 1$ distances, the induced metric on V' must have at least $n - 1 - (n - n') = n' - 1$ distances.

Now denote by k the number of internal nodes in $T_{V'}$, and let us show that $k = n' - 1$. In one direction, $k \geq n' - 1$ because by the above claim, the tree $T_{V'}$ must have at least $n' - 1$ distinct labels. For the other direction we count degrees. Every internal node in $T_{V'}$ has at least two children, every internal node has degree at least 3, except for the root which has at least 2, hence the sum of degrees in $T_{V'}$ is at least $n' + 3k - 1$. At the same time, $T_{V'}$ is a tree and has exactly $n' + k - 1$ edges, hence this sum of degrees is $2(n' + k - 1) \geq n' + 3k - 1$,

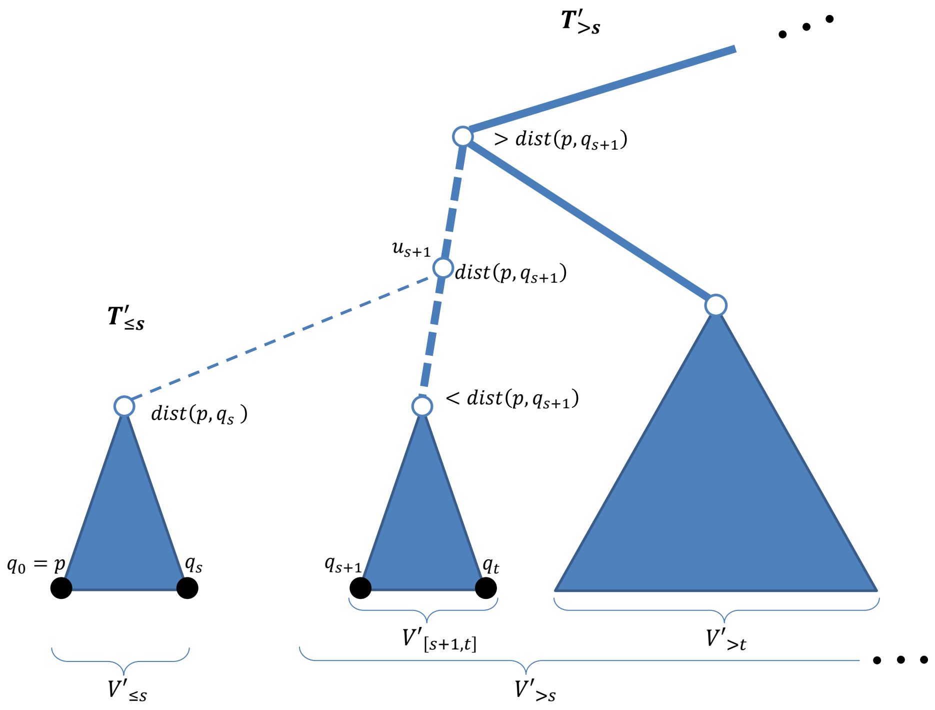

Figure 5.6: An illustration of the algorithm. Bold lines represent edges in $T'_{>s}$, and dashed lines represent edges affected by connecting $T'_{\leq s}$ into this tree.

i.e., $k \leq n' - 1$. We conclude that both inequalities above hold with equality, which implies that all $k = n' - 1$ internal nodes have distinct labels, and none of them can have three or more children. $\square$

Continuing with the proof of Theorem 5.5.3, let us now prove that the tree $T'_{V'}$ constructed by the algorithm represents all the distances correctly. It suffices to consider $u \in T'_{\leq s}$ and $v \in T'_{>s}$, and show that in the combined tree

$$\text{label}_{T'_{V'}}(u, v) = \text{dist}(u, v).$$

By the ordering of V', we have $\text{dist}(u, p) \leq \text{dist}(q_s, p) < \text{dist}(q_{s+1}, p) \leq \text{dist}(v, p)$, and thus by (5.3), $\text{dist}(u, v) = \text{dist}(v, p)$. Since both $u, p \in T'_{\leq s}$, we have $\text{label}_{T_{V'}}(v, p) = \text{label}_{T_{V'}}(u, v)$, and thus it suffices to show that

$$\text{label}_{T_{V'}}(v, p) = \text{dist}(v, p).$$

We now have two case, as follows. Let $t \geq s + 1$ be the largest such that $\text{dist}(q_t, p) = \text{dist}(q_{s+1}, p)$, and partition $V'_{>s}$ into $V'_{[s+1,t]} = \{q_{s+1}, \dots, q_t\}$ and (possibly empty) $V'_{>t} = \{q_{t+1}, \dots, q_{n'}\}$. Suppose first that $v \in V'_{[s+1,t]}$. In this case, by the way we connected the two trees, the LCA of p and v is the same as of p and q_{s+1} (i.e., the new node u_{s+1}), and thus $\text{label}_{T'_{V'}}(v, p) = \text{dist}(q_{s+1}, p) = \text{dist}(v, p)$, as required. Suppose next that $v \in V'_{>t}$. In this case, we shall show $\text{label}_{T'_{V'}}(v, p) = \text{dist}(v, q_{s+1}) = \text{dist}(v, p)$. The first equality is because by the way we connected the two trees, the LCA of p and v is the same as of q_{s+1} and v.

For the second equality, observe that $\mathrm{dist}(q_{s+1}, p) < \mathrm{dist}(q_{s+1}, v)$ by inspecting at the LCA of each pair, and now use (5.3) on the "triangle" p, q_{s+1}, v to identify its two largest distances as $\mathrm{dist}(v, q_{s+1}) = \mathrm{dist}(v, p)$. We conclude that indeed in all cases $\mathrm{label}_{T_{V'}}(v, p) = \mathrm{dist}(v, p)$.

We proceed to show that with high probability, the algorithm makes only $O(n \log n)$ distance queries. We first claim that for every $V' \subseteq V$ (and thus every instance throughout the recursion), every representing tree $T_{V'}$ has a centroid-like node c^*, where the number of leaves under it in the tree $T_{V'}$ is in the range $[\lceil n'/4 \rceil, \lceil n'/2 \rceil]$. To see this, start with the root of $T_{V'}$, and follow the child with more leaves under it, until that number is no larger than $\lceil n'/2 \rceil$. Because the tree is binary by Claim 5.5.4, this stops at a node c^* where the number of leaves under it is some $s^* \in [\lceil n'/4 \rceil, \lceil n'/2 \rceil]$, as claimed. Now, a uniformly random pivot p has probability $s^*/n' \geq 1/4$ to be a descendant of c^*, in which case (5.4) holds. Thus (5.4) occurs with probability at least $1/4$.

Consider now an execution of the algorithm, and describe it using a recursion tree defined as follows (note the difference from a representing tree of V'). In this tree, a vertex (we use this term to distinguish from the nodes in the trees discussed above) corresponds to an instance of the recursion and has two children corresponding to the two new instances if a successful pivot is picked, and has one child if an unsuccessful pivot is picked. Thus, this recursion tree has a vertex for every pivot that is picked. The total number of distance queries performed at each depth i in the recursion tree is bounded by n, because instances at the same depth i have pairwise-disjoint node sets, and every instance performs exactly one query for every non-pivot node (for its distance to the pivot in the same instance). It thus suffice to show that with high probability, the depth of the recursion tree is at most $8 \log_{4/3} n$, and this would imply that the total number of queries is $O(n \log n)$. To see end, fix a node $j \in V$; its root-to-leaf path in the recursion tree contains at most $\log_{4/3} n$ successful pivots, as these already reduce the instance size to at most 1. Now imagine these random pivots an infinite sequence of coins with probability of success (heads) at least $1/4$, even when conditioned on the outcomes of earlier coins. With probability at least $1 - 1/n^2$, the prefix of $16 \log_{4/3} n$ first random coins already contains at least $\log_{4/3} n$ heads. If this high-probability event occurs, there are enough successful pivots (heads) to guarantee that the recursion terminates before that coins prefix is exhausted, which means that node j goes through at most $16 \log_{4/3} n$ pivots. By union bound over all n nodes, we conclude that with high probability the depth of the recursion tree is at most $16 \log_{4/3} n$, in which case the total number of distance queries is $O(n \log n)$. Finally, we bound the sorting of the distances the algorithm does for each instance from the pivot in order to check if (5.4) holds. This takes $c \cdot n' \log n'$ for some constanct c by a standard sorting algorithm, and by using the recursion tree as before, the sorting for all instances at a single depth j takes time $\sum_{V_i' \in depth j} c \cdot n_i \log n_i \leq O(n \log n)$, where $|V_i'| = n_i$, and the inequality is by the convexity of $n_i \log n_i$. Then, multiply by the height of the recursion tree $O(\log n)$ to get the term $O(n \log^2 n)$. Note that connecting the trees that came back from the recursion takes $O(\log n')$ time, which is much smaller than the sorting and thus is bounded as well. Altogether, we get a total running time of $O(n \log n Q(n) + n \log^2 n)$, as required. This concludes Theorem 5.5.3. □

Bibliography

[ABBK17] A. Abboud, A. Backurs, K. Bringmann, and M. Künnemann. Fine-grained complexity of analyzing compressed data: Quantifying improvements over decompress-and-solve. In *FOCS*, pages 192–203, 2017. doi:10.1109/FOCS.2017.12.

[ABDN18] A. Abboud, K. Bringmann, H. Dell, and J. Nederlof. More consequences of falsifying SETH and the orthogonal vectors conjecture. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018*, pages 253–266, 2018. doi:10.1145/3188745.3188938.

[ABHS17] A. Abboud, K. Bringmann, D. Hermelin, and D. Shabtay. SETH-based lower bounds for subset sum and bicriteria path. *CoRR*, 2017. Available from: http://arxiv.org/abs/1704.04546.

[ABMR11] E. Alkassar, S. Böhme, K. Mehlhorn, and C. Rizkallah. Verification of certifying computations. In *International Conference on Computer Aided Verification*, pages 67–82. Springer, 2011.

[ABW15] A. Abboud, A. Backurs, and V. V. Williams. Tight hardness results for LCS and other sequence similarity measures. In *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015*, pages 59–78, 2015. doi:10.1109/FOCS.2015.14.

[ABW18] A. Abboud, A. Backurs, and V. V. Williams. If the current clique algorithms are optimal, so is Valiant's parser. *SIAM J. Comput.*, 47(6):2527–2555, 2018. doi:10.1137/16M1061771.

[ACLY00] R. Ahlswede, N. Cai, S. R. Li, and R. W. Yeung. Network information flow. *IEEE Trans. Information Theory*, 46(4):1204–1216, 2000. doi:10.1109/18.850663.

[ACZ98] S. R. Arikati, S. Chaudhuri, and C. D. Zaroliagis. All-pairs min-cut in sparse networks. *J. Algorithms*, 29(1):82–110, 1998.

[AGI$^+$19] A. Abboud, L. Georgiadis, G. F. Italiano, R. Krauthgamer, N. Parotsidis, O. Trabelsi, P. Uznanski, and D. Wolleb-Graf. Faster Algorithms for All-Pairs Bounded Min-Cuts. In *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*, volume 132, pages 7:1–7:15, 2019. doi:10.4230/LIPIcs.ICALP.2019.7.

[AHK12] S. Arora, E. Hazan, and S. Kale. The multiplicative weights update method: a meta-algorithm and applications. *Theory of Computing*, 8(1):121–164, 2012. doi:10.4086/toc.2012.v008a006.

[AKT20a] A. Abboud, R. Krauthgamer, and O. Trabelsi. Cut-equivalent trees are optimal for min-cut queries in undirected graphs. *Accepted to FOCS'20*, 2020.

[AKT20b] A. Abboud, R. Krauthgamer, and O. Trabelsi. New algorithms and lower bounds for all-pairs max-flow in undirected graphs. In *Proceedings of the Thirty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '20, page 48–61, USA, 2020. doi:10.1137/1.9781611975994.4.

[AMO93] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network flows - theory, algorithms and applications.* Prentice Hall, 1993.

[AV18] N. Anari and V. V. Vazirani. Planar graph perfect matching is in NC. In M. Thorup, editor, *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018*, pages 650–661, 2018. `doi:10.1109/FOCS.2018.00068`.

[AW14] A. Abboud and V. V. Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014*, pages 434–443, 2014. `doi:10.1109/FOCS.2014.53`.

[AWY15] A. Abboud, R. Williams, and H. Yu. More applications of the polynomial method to algorithm design. In *Proceedings of the Twenty-sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '15, pages 218–230, 2015. `doi:10.1145/2722129.2722146`.

[AWY18] A. Abboud, V. V. Williams, and H. Yu. Matching triangles and basing hardness on an extremely popular conjecture. *SIAM J. Comput.*, 47(3):1098–1122, 2018. `doi:10.1137/15M1050987`.

[BCH$^+$08] A. Bhalgat, R. Cole, R. Hariharan, T. Kavitha, and D. Panigrahi. Efficient algorithms for Steiner edge connectivity computationand Gomory-Hu tree construction for unweighted graphs. Unpublished full version of [BHKP07], 2008. Available from: `http://hariharan-ramesh.com/papers/gohu.pdf`.

[BENW16] G. Borradaile, D. Eppstein, A. Nayyeri, and C. Wulff-Nilsen. All-pairs minimum cuts in near-linear time for surface-embedded graphs. In *32nd International Symposium on Computational Geometry, SoCG 2016*, pages 22:1–22:16, 2016.

[BFJ95] J. Bang-Jensen, A. Frank, and B. Jackson. Preserving and increasing local edge-connectivity in mixed graphs. *SIAM J. Discret. Math.*, 8(2):155–178, 1995. `doi:10.1137/S0036142993226983`.

[BGL17] K. Bringmann, A. Grønlund, and K. G. Larsen. A dichotomy for regular expression membership testing. In *FOCS*, pages 307–318, 2017. `doi:10.1109/FOCS.2017.36`.

[BHKP07] A. Bhalgat, R. Hariharan, T. Kavitha, and D. Panigrahi. An $O(mn)$ Gomory-Hu tree construction algorithm for unweighted graphs. In *39th Annual ACM Symposium on Theory of Computing*, STOC'07, pages 605–614. ACM, 2007. `doi:10.1145/1250790.1250879`.

[BJS10] M. S. Bazaraa, J. J. Jarvis, and H. D. Sherali. *Linear Programming and Network Flows.* John Wiley & Sons, Inc., Hoboken, NJ, fourth edition, 2010.

[BK09] G. Borradaile and P. Klein. An $\tilde{O}(n \log n)$ algorithm for maximum *st*-flow in a directed planar graph. *J. ACM*, 56(2):9:1–9:30, April 2009. `doi:10.1145/1502793.1502798`.

[BK15a] A. A. Benczúr and D. R. Karger. Randomized approximation schemes for cuts and flows in capacitated graphs. *SIAM J. Comput.*, 44(2):290–319, 2015. `doi:10.1137/070705970`.

[BK15b] K. Bringmann and M. Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In V. Guruswami, editor, *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015*, pages 79–97, 2015. `doi:10.1109/FOCS.2015.15`.

[BSW15] G. Borradaile, P. Sankowski, and C. Wulff-Nilsen. Min *st*-cut oracle for planar graphs with near-linear preprocessing time. *ACM Trans. Algorithms*, 11(3), 2015. `doi:10.1145/2684068`.

[BW17] K. Bringmann and P. Wellnitz. Clique-based lower bounds for parsing tree-adjoining grammars. In *CPM*, pages 12:1–12:14, 2017. `doi:10.4230/LIPIcs.CPM.2017.12`.

[CGI+16] M. L. Carmosino, J. Gao, R. Impagliazzo, I. Mihajlin, R. Paturi, and S. Schneider. Nondeterministic extensions of the strong exponential time hypothesis and consequences for non-reducibility. In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science*, ITCS '16, pages 261–270. ACM, 2016. doi:10.1145/2840728.2840746.

[CH03] R. Cole and R. Hariharan. A fast algorithm for computing steiner edge connectivity. In *Proceedings of the Thirty-fifth Annual ACM Symposium on Theory of Computing*, STOC '03, pages 167–176. ACM, 2003. doi:10.1145/780542.780568.

[Cha15] Y. Chang. Conditional lower bound for RNA folding problem. *CoRR*, abs/1511.04731, 2015. arXiv:1511.04731.

[Chi60] R. T. Chien. Synthesis of a communication net. *IBM Journal of Research and Development*, 4(3):311–320, 1960.

[CLL13] H. Y. Cheung, L. C. Lau, and K. M. Leung. Graph connectivities, network coding, and expander graphs. *SIAM Journal on Computing*, 42(3):733–751, 2013. doi:10.1137/110844970.

[CW90] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, 1990. doi:10.1016/S0747-7171(08)80013-2.

[CW16] T. M. Chan and R. Williams. Deterministic apsp, orthogonal vectors, and more: Quickly derandomizing razborov-smolensky. In *Proceedings of the Twenty-seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '16, pages 1246–1255, 2016. doi:10.1145/2884435.2884522.

[Edm70] J. Edmonds. Submodular functions, matroids, and certain polyhedra. *Combinatorial structures and their applications*, pages 69–87, 1970.

[EG04] F. Eisenbrand and F. Grandoni. On the complexity of fixed parameter clique and dominating set. *Theor. Comput. Sci.*, 326(1-3):57–67, 2004. doi:10.1016/j.tcs.2004.05.009.

[FF56] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian journal of Mathematics*, 8(3):399–404, 1956. Available from: http://www.rand.org/pubs/papers/P605/.

[FF62] L. R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks.* Princeton University Press, 1962.

[FKT19] A. Filtser, R. Krauthgamer, and O. Trabelsi. Relaxed voronoi: A simple framework for terminal-clustering problems. In *2nd Symposium on Simplicity in Algorithms, SOSA@SODA 2019, January 8-9, 2019 - San Diego, CA, USA*, volume 69, pages 10:1–10:14, 2019. doi:10.4230/OASIcs.SOSA.2019.10.

[Fre95] G. N. Frederickson. Using cellular graph embeddings in solving all pairs shortest paths problems. *J. Algorithms*, 19(1):45–85, July 1995. doi:10.1006/jagm.1995.1027.

[Gab95] H. N. Gabow. A matroid approach to finding edge connectivity and packing arborescences. *J. Comput. Syst. Sci.*, 50(2):259–273, 1995.

[GGI+17] L. Georgiadis, D. Graf, G. F. Italiano, N. Parotsidis, and P. Uznanski. All-pairs 2-reachability in o(n^w log n) time. In I. Chatzigiannakis, P. Indyk, F. Kuhn, and A. Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, volume 80 of *LIPIcs*, pages 74:1–74:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. Available from: https://doi.org/10.4230/LIPIcs.ICALP.2017.74, doi:10.4230/LIPIcs.ICALP.2017.74.

[GH61] R. E. Gomory and T. C. Hu. Multi-terminal network flows. *Journal of the Society for Industrial and Applied Mathematics*, 9:551–570, 1961. Available from: `http://www.jstor.org/stable/2098881`.

[GH86] F. Granot and R. Hassin. Multi-terminal maximum flows in node-capacitated networks. *Discrete Applied Mathematics*, 13(2-3):157–163, 1986.

[GIKW17] J. Gao, R. Impagliazzo, A. Kolokolova, and R. R. Williams. Completeness for first-order properties on sparse structures with algorithmic applications. In *28th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017*, pages 2162–2181, 2017. `doi:10.1137/1.9781611974782.141`.

[GT01] A. V. Goldberg and K. Tsioutsiouliklis. Cut tree algorithms: an experimental study. *Journal of Algorithms*, 38(1):51–83, 2001.

[Gus90] D. Gusfield. Very simple methods for all pairs network flow analysis. *SIAM Journal on Computing*, 19(1):143–155, 1990.

[GV12] V. Gurvich and M. N. Vyalyi. Characterizing (quasi-)ultrametric finite spaces in terms of (directed) graphs. *Discret. Appl. Math.*, 160(12):1742–1756, 2012. `doi:10.1016/j.dam.2012.03.034`.

[HK95] M. R. Henzinger and V. King. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing*, page 519–527, 1995. `doi:10.1145/225058.225269`.

[HKNR98] T. Hagerup, J. Katajainen, N. Nishimura, and P. Ragde. Characterizing multiterminal flow networks and computing flows in networks of small treewidth. *J. Comput. Syst. Sci.*, 57:366–375, 1998. `doi:10.1006/jcss.1998.1592`.

[HKP07] R. Hariharan, T. Kavitha, and D. Panigrahi. Efficient algorithms for computing all low $s-t$ edge connectivities and related problems. In *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 127–136. SIAM, 2007. Available from: `http://dl.acm.org/citation.cfm?id=1283383.1283398`.

[HL07] R. Hassin and A. Levin. Flow trees for vertex-capacitated networks. *Discrete Appl. Math.*, 155(4):572–578, 2007. `doi:10.1016/j.dam.2006.08.012`.

[HO94] J. Hao and J. B. Orlin. A faster algorithm for finding the minimum cut in a directed graph. *J. Algorithms*, 17(3):424–446, 1994. `doi:10.1006/jagm.1994.1043`.

[IP01] R. Impagliazzo and R. Paturi. On the complexity of k-SAT. *Journal of Computer and System Sciences*, 62(2):367–375, March 2001. `doi:10.1006/jcss.2000.1727`.

[IPZ01] R. Impagliazzo, R. Paturi, and F. Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, 63(4):512–530, December 2001. `doi:10.1006/jcss.2001.1774`.

[Jel63] F. Jelinek. On the maximum number of different entries in the terminal capacity matrix of oriented communication nets. *IEEE Transactions on Circuit Theory*, 10(2):307–308, 1963. `doi:10.1109/TCT.1963.1082149`.

[KL15] D. R. Karger and M. S. Levine. Fast augmenting paths by random sampling from residual graphs. *SIAM J. Comput.*, 44(2):320–339, 2015. `doi:10.1137/070705994`.

[KLOS14] J. A. Kelner, Y. T. Lee, L. Orecchia, and A. Sidford. An almost-linear-time algorithm for approximate max flow in undirected graphs, and its multicommodity generalizations. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014*, pages 217–226, 2014. `doi:10.1137/1.9781611973402.16`.

[KM03] R. Koetter and M. Médard. An algebraic approach to network coding. *IEEE/ACM Trans. Netw.*, 11(5):782–795, 2003. `doi:10.1109/TNET.2003.818197`.

[KT18a] L. Kamma and O. Trabelsi. Nearly optimal time bounds for kpath in hypergraphs. *arXiv preprint arXiv:1803.04940*, 2018.

[KT18b] R. Krauthgamer and O. Trabelsi. Conditional lower bounds for all-pairs max-flow. *ACM Trans. Algorithms*, 14(4):42:1–42:15, 2018. `doi:10.1145/3212510`.

[KT19] R. Krauthgamer and O. Trabelsi. The set cover conjecture and subgraph isomorphism with a tree pattern. In *36th International Symposium on Theoretical Aspects of Computer Science, STACS 2019*, pages 45:1–45:15, 2019. `doi:10.4230/LIPIcs.STACS.2019.45`.

[Kün18] M. Künnemann. On nondeterministic derandomization of Freivalds' algorithm: Consequences, avenues and algorithmic progress. In *26th Annual European Symposium on Algorithms, ESA 2018, August 20-22, 2018, Helsinki, Finland*, pages 56:1–56:16, 2018. `doi:10.4230/LIPIcs.ESA.2018.56`.

[LG14] F. Le Gall. Powers of tensors and fast matrix multiplication. In *ISSAC*, pages 296–303, 2014. `doi:10.1145/2608628.2608664`.

[LNSW12] J. Lacki, Y. Nussbaum, P. Sankowski, and C. Wulff-Nilsen. Single source - all sinks max flows in planar digraphs. In *53rd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2012*, pages 599–608, 2012. `doi:10.1109/FOCS.2012.66`.

[LS14] Y. T. Lee and A. Sidford. Path finding methods for linear programming: Solving linear programs in õ(vrank) iterations and faster algorithms for maximum flow. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014*, pages 424–433, 2014. `doi:10.1109/FOCS.2014.52`.

[LS20a] Y. P. Liu and A. Sidford. Faster energy maximization for faster maximum flow. In *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 803–814, 2020. `doi:10.1145/3357713.3384247`.

[LS20b] Y. P. Liu and A. Sidford. Unit capacity maxflow in almost m time. *CoRR*, abs/2003.08929, 2020. Available from: `https://arxiv.org/abs/2003.08929`, `arXiv:2003.08929`.

[LWW18] A. Lincoln, V. V. Williams, and R. R. Williams. Tight hardness for shortest cycles and paths in sparse graphs. In *SODA*, pages 1236–1252, 2018.

[LYC03] S.-Y. Li, R. W. Yeung, and N. Cai. Linear network coding. *IEEE transactions on information theory*, 49(2):371–381, 2003. `doi:10.1109/TIT.2002.807285`.

[Mad16] A. Madry. Computing maximum flow with augmenting electrical flows. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016*, pages 593–602, 2016. `doi:10.1109/FOCS.2016.70`.

[May60] W. Mayeda. Terminal and branch capacity matrices of a communication net. *IRE Transactions on Circuit Theory*, 7(3):261–269, 1960. `doi:10.1109/TCT.1960.1086673`.

[May62] W. Mayeda. On oriented communication nets. *IRE Transactions on Circuit Theory*, 9(3):261–267, 1962. `doi:10.1109/TCT.1962.1086912`.

[Men27] K. Menger. Zur allgemeinen kurventheorie. *Fundamenta Mathematicae*, 10(1):96–115, 1927.

[MMNS11] R. M. McConnell, K. Mehlhorn, S. Näher, and P. Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, 2011.

[MVV87] K. Mulmuley, U. V. Vazirani, and V. V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7(1):105–113, 1987. `doi:10.1007/BF02579206`.

[NP85] J. Nešetřil and S. Poljak. On the complexity of the subgraph problem. *Commentationes Mathematicae Universitatis Carolinae*, 26(2):415–419, 1985.

[Pan16] D. Panigrahi. Gomory-Hu trees. In M.-Y. Kao, editor, *Encyclopedia of Algorithms*, pages 858–861. Springer, 2016. doi:10.1007/978-1-4939-2864-4.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, 1985.

[Räc02] H. Räcke. Minimizing congestion in general networks. In *43rd Symposium on Foundations of Computer Science, FOCS 2002*, pages 43–52, 2002. doi:10.1109/SFCS.2002.1181881.

[RST14] H. Räcke, C. Shah, and H. Täubig. Computing cut-based hierarchical decompositions in almost linear time. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '14, page 227–238. SIAM, 2014. doi:10.1137/1.9781611973402.17.

[Sch02] A. Schrijver. On the history of the transportation and maximum flow problems. *Math. Program.*, 91(3):437–445, 2002. doi:10.1007/s101070100259.

[ST18] A. Sidford and K. Tian. Coordinate methods for accelerating ℓ_∞ regression and faster approximate maximum flow. In *FOCS '18*, pages 922–933. IEEE Computer Society, 2018. doi:10.1109/FOCS.2018.00091.

[Vas09] V. Vassilevska. Efficient algorithms for clique problems. *Inf. Process. Lett.*, 109(4):254–257, 2009. doi:10.1016/j.ipl.2008.10.014.

[Vas12] V. Vassilevska Williams. Multiplying matrices faster than Coppersmith-Winograd. In *STOC*, pages 887–898, 2012. doi:10.1145/2213977.2214056.

[Vas15] V. Vassilevska-Williams. Hardness of Easy Problems: Basing Hardness on Popular Conjectures such as the Strong Exponential Time Hypothesis (Invited Talk). In *10th International Symposium on Parameterized and Exact Computation (IPEC 2015)*, volume 43 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17–29. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015. doi:10.4230/LIPIcs.IPEC.2015.17.

[Vas18] V. Vassilevska-Williams. On some fine-grained questions in algorithms and complexity. In *Proceedings of ICM*, 2018. To Appear. Available from: http://people.csail.mit.edu/virgi/eccentri.pdf.

[Wil05] R. Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theoretical Computer Science*, 348(2):357–365, 2005. doi:10.1016/j.tcs.2005.09.023.

[Wil16] R. R. Williams. Strong ETH breaks with Merlin and Arthur: Short non-interactive proofs of batch evaluation. In *31st Conference on Computational Complexity, CCC 2016*, pages 2:1–2:17, 2016.

[WL93] Z. Wu and R. Leahy. An optimal graph theoretic approach to data clustering: Theory and its application to image segmentation. *IEEE transactions on pattern analysis and machine intelligence*, 15(11):1101–1113, 1993.

[WW18] V. V. Williams and R. R. Williams. Subcubic equivalences between path, matrix, and triangle problems. *J. ACM*, 65(5):27:1–27:38, 2018. doi:10.1145/3186893.

[Yu18] H. Yu. An improved combinatorial algorithm for Boolean matrix multiplication. *Inf. Comput.*, 261:240–247, 2018. doi:10.1016/j.ic.2018.02.006.

Appendix A

Papers not Included in the Thesis

The following are the abstract sections of papers that were written during my PhD period but are not included in this thesis.

Relaxed Voronoi: A Simple Framework for Terminal-Clustering Problems [FKT19]

We reprove three known algorithmic bounds for terminal-clustering problems, using a single framework that leads to simpler proofs. In this genre of problems, the input is a metric space (X, d) (possibly arising from a graph) and a subset of terminals $K \subset X$, and the goal is to partition the points X such that each part, called a cluster, contains exactly one terminal (possibly with connectivity requirements) so as to minimize some objective. The three bounds we reprove are for Steiner Point Removal on trees [Gupta, SODA 2001], for Metric 0-Extension in bounded doubling dimension [Lee and Naor, unpublished 2003], and for Connected Metric 0-Extension [Englert et al., SICOMP 2014].

A natural approach is to cluster each point with its closest terminal, which would partition X into so-called Voronoi cells, but this approach can fail miserably due to its stringent cluster boundaries. A now-standard fix, which we call the `Relaxed-Voronoi` framework, is to use enlarged Voronoi cells, but to obtain disjoint clusters, the cells are computed greedily according to some order. This method, first proposed by Calinescu, Karloff and Rabani [SICOMP 2004], was employed successfully to provide state-of-the-art results for terminal-clustering problems on general metrics. However, for restricted families of metrics, e.g., trees and doubling metrics, only more complicated, ad-hoc algorithms are known. Our main contribution is to demonstrate that the `Relaxed-Voronoi` algorithm is applicable to restricted metrics, and actually leads to relatively simple algorithms and analyses.

The Set Cover Conjecture and Subgraph Isomorphism with a Tree Pattern [KT19]

In the Set Cover problem, the input is a ground set of n elements and a collection of m sets, and the goal is to find the smallest sub-collection of sets whose union is the entire ground set. The fastest algorithm known runs in time $O(mn2^n)$ [Fomin et al., WG 2004], and the Set Cover Conjecture (SeCoCo) [Cygan et al., TALG 2016] asserts that for every fixed $\varepsilon > 0$, no algorithm can solve Set Cover in time $2^{(1-\varepsilon)n}\operatorname{poly}(m)$, even if set sizes are bounded by $\Delta = \Delta(\varepsilon)$. We show strong connections between this problem and k-Tree, a special case of Subgraph Isomorphism where the input is an n-node graph G and a k-node tree T, and the goal is to determine whether G has a subgraph isomorphic to T.

First, we propose a weaker conjecture Log-SeCoCo, that allows input sets of size $\Delta = O(1/\varepsilon \cdot \log n)$, and show that an algorithm breaking Log-SeCoCo would imply a faster algorithm than the currently known $2^n\operatorname{poly}(n)$-time algorithm [Koutis and Williams, TALG 2016] for Directed nTree, which is k-Tree with $k = n$ and arbitrary directions to the edges of G and T. This would also improve the running time for Directed Hamiltonicity, for which no algorithm significantly faster than $2^n\operatorname{poly}(n)$ is known despite extensive research.

Second, we prove that if Set Cover cannot be solved significantly faster than $2^n\operatorname{poly}(m)$ (an assumption even weaker than Log-SeCoCo), then k-Tree cannot be computed significantly faster than $2^k\operatorname{poly}(n)$, the running time of the Koutis and Williams' algorithm. Applying the same techniques to the p-Partial Cover problem, a parameterized version of Set Cover that requires covering at least p elements, we obtain a new algorithm with running time $(2+\varepsilon)^p(m+n)^{O(1/\varepsilon)}$ for arbitrary $\varepsilon > 0$, which improves previous work and is nearly optimal assuming SeCoCo.

Nearly Optimal Time Bounds for k-Path in Hypergraphs [KT18a]

We study the k-HyperPath problem: Given an r-uniform hypergraph for some integer r, the goal is to find a tight path of length k, that is, a sequence of k nodes such that every consecutive r of them constitute a hyperedge in the graph. This problem is a natural generalization of k-Path in graphs, and was investigated for large values of the uniformity parameter $r = \Omega(k)$ (Lincoln, V.Williams, and Williams, SODA 2018), where a conditional lower bound of $\tilde{O}(n^k)$ was presented (where throughout, n is the number of nodes and m is the number of (hyper)edges).

We give almost tight conditional lower bounds on the running time of the k-HyperPath problem for smaller values of r, showing that an algorithm with running time $O^*(2^{(1-\gamma)k})$ where $\gamma > 0$ is independent of r is probably impossible. Specifically, it implies that Set Cover on n elements can be solved in time $O^*(2^{(1-\delta)n})$ for some $\delta > 0$. To complete the picture, we show that a known algorithm for k-Path (Koutis and Williams, TALG 2016) could be extended to k-HyperPath for every integer $r \geq 3$, with running time $2^k m \cdot \operatorname{poly}(n)$. We believe a by-product of our conditional lower bound techniques is diversifying the scope of reductions from Set Cover.