Preconditioning and Locality in Algorithm Design

Jason Li

CMU-CS-21-119

June 2021

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Anupam Gupta, Chair
Bernhard Haeupler, Chair
Gary Miller
Satish Rao (UC Berkeley)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by National Science Foundation award numbers: CCF-1536002, CCF-1540541, CCF-1617790, CCF-1907820, CCF-1955785, and CCF-2006953. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity

Keywords: preconditioning, locality, graph algorithms, minimum cut, expander graphs

To my family

Abstract

Algorithms is a broad, rich, and fast-growing field. For the latter half of last century, many branches of algorithms have emerged and grown in popularity, and many different techniques have been invented to solve the central problems in each area. Some of these techniques, such as the *push-relabel* algorithm for maximum flow, are specially designed to solve a single problem. Other techniques, such as the multiplicative weights update method, are more general and applicable to a wide range of problems. And others, such as dynamic programming, divide and conquer, and linear programming relaxation and rounding, are so fundamental that they have not only pervaded every branch of algorithms, but have ultimately reshaped the way we approach algorithm design.

This thesis is devoted to studying two more modern algorithmic techniques, namely *preconditioning* and *locality*, which were pioneered by Spielman and Teng [100] in their ground-breaking work on Laplacian system solvers and have seen countless new applications in the past decade. In this thesis, I successfully apply preconditioning and locality to resolve fundamental open problems from a wide array of algorithmic subfields, from fast, sequential algorithms to deterministic algorithms to parallel algorithms, thereby demonstrating the power and versatility of the two techniques. Taking one step further, I make my case that preconditioning and locality are more than just powerful tools with countless applications: they are new, fundamental ways of thinking about algorithms that have the potential to revolutionize algorithm design just like dynamic programming and divide and conquer had done in the past.

Acknowledgements

First and foremost, I would like to thank my advisors, Anupam Gupta and Bernhard Haeupler, for all the inspiration and support they have provided over the years. Research was not always as fruitful or rewarding as it is now, and I am forever grateful to have had such caring and supportive advisors to guide my way through the early, most formative years of my PhD.

Second, I would like to thank my other thesis committee members, Gary Miller and Satish Rao. Thank you both for all the fun, inspiring discussions we have had, and I am honored to have such esteemed and renowned researchers on my thesis committee.

Research is a collaborative process of mutual learning and discovery, and I would like to thank all of my collaborators throughout my PhD journey without whom research would not be the same. Special thanks to my closest collaborator, Thatchaphol Saranurak, for all our productive, late-night research sessions, leading to a total of six papers over the two years we have worked together. I would also like to thank Nikhil Bansal, Julia Chuzhoy, Vincent Cohen-Addad, Michael Elkin, Mohsen Ghaffari, Janardhan Kulkarni, Danupon Nanongkai, Jesper Nederlof, Ofer Neiman, Debmalya Panigrahi, and Merav Parter for generously hosting me during my many research visits, as well as all of my other collaborators: John Augustine, Yu Gao, Robert Gmyr, Zhiyang He, Ellis Hershkowitz, Kristian Hinnenthal, Philip Klein, Stefan Kratsch, Fabian Kuhn, Amit Kumar, Euiwoong Lee, Pasin Manurangsi, Richard Peng, Christian Scheideler, Magnus Wahlstrom, Michal Wlodarczyk, Sorrachai Yingchareonthawornchai.

Next, I would like to thank all the fellow graduate students I have had the pleasure of meeting, for no journey is complete without the friends we make along the way. Special thanks to my first year office mates Timothy Chu and Goran Zuzic for all the wacky stuff we did together, from sharing cool theorems to playing foosball to making memes, and to Guru Guruganesh, Euiwoong Lee, and David Wajc as the senior PhD students I looked up to the most. I also recall fun times with Dan Anderson, Ainesh Bakshi, Vijay Bhattiprolu, Laxman Dhulipala, Bailey Flanigan, Isaac Grosof, Ellis Hershkowitz, Raj Jayaram, Pedro Paredes, Nicolas Resch, Ziv Scully, Sahil Singla, Ameya Velingker, David Witmer, John Wright, Xinyu Wu, and Yu Zhao.

Last but not least, to restrict my acknowledgements to the duration of my PhD would be tantamount to skipping the prologue to my academic journey: why I pursued a PhD in the first place! I was fortunate enough to have had supportive, influential figures since middle school, from my eighth grade math contest coach Joshua Frost who was one of the most passionate and driven teachers I have known, to my high school PRIMES mentors Pavel Etingof and David Jordan who introduced me to the wonderful world of academic research, and to Ryan O'Donnell and Anupam again for their amazing, inspirational theoretical computer science classes I took as an undergraduate at CMU which convinced me to stay for my PhD. And finally, my greatest thanks goes to none other than my parents and my brother David who have supported me, both emotionally and financially, at every stage of my twenty long years of education.

Contents

1 Introduction **7**
1.1 Locality: Unbalanced vs. Balanced . . . 8
1.1.1 Minimum Isolating Cuts and Applications . . . 10
1.2 Preconditioning: Worst Case vs. Average Case . . . 12
1.2.1 Graph Cut Problems . . . 14
1.2.2 Graph Distance Problems . . . 15
1.3 Preliminaries . . . 16
1.4 Bibliographic Notes . . . 17

I Locality **19**

2 Minimum Isolating Cuts **21**
2.1 Background . . . 21
2.2 The Isolating Cuts Algorithm . . . 22
2.3 Conclusion . . . 24

3 Steiner Mincut **27**
3.1 Background . . . 28
3.2 Randomized Steiner Mincut . . . 28
3.3 Deterministic Steiner Mincut . . . 29
3.3.1 Unbalanced Case . . . 29
3.3.2 Balanced Case: Sparsifying U . . . 32
3.4 Conclusion . . . 37

4 Gomory-Hu Tree **39**
4.1 Background . . . 40
4.2 Our Results . . . 40
4.3 Our Techniques . . . 42
4.4 Additional Preliminaries . . . 45
4.5 Reducing to SSMC Verification . . . 46
4.5.1 A Single Recursive Step . . . 47

4.5.2 The Gomory-Hu Tree Algorithm . . . 49
4.6 The Cut Threshold Algorithm . . . 53
4.7 Approximate GH Tree . . . 56
4.7.1 Approximation . . . 56
4.7.2 Running Time Bound . . . 60
4.7.3 Weighted Graphs . . . 64
4.8 Conclusion . . . 65

5 Directed Global Mincut **67**
5.1 Background . . . 67
5.2 Our Techniques . . . 68
5.2.1 Additional Preliminaries . . . 69
5.3 The Directed Mincut Algorithm . . . 70
5.4 Sparsification . . . 71
5.5 Finding a 1-respecting Arborescence . . . 73
5.6 Mincut Given 1-respecting Arborescence . . . 75
5.7 Conclusion . . . 78

II Preconditioning **81**

6 Deterministic Mincut **83**
6.1 Background . . . 83
6.2 Our Techniques . . . 84
6.3 Additional Preliminaries . . . 87
6.3.1 Karger's Approach . . . 87
6.3.2 Spectral Graph Theory . . . 88
6.4 Expander Case . . . 89
6.5 General Case . . . 92
6.5.1 Expander Decomposition Preliminaries . . . 93
6.5.2 Unbalanced Case . . . 96
6.5.3 Balanced Case . . . 103
6.5.4 Combining Them Together . . . 104
6.5.5 Removing the Maximum Weight Assumption . . . 106
6.6 Conclusion . . . 107

7 Parallel Shortest Path **109**
7.1 Background . . . 110
7.1.1 Our Contributions . . . 110
7.1.2 Our Techniques . . . 111
7.1.3 Chapter Organization . . . 114

7.2 Additional Preliminaries 114
7.2.1 PRAM Model 114
7.2.2 Transshipment Preliminaries 114
7.2.3 Parallel Shortest Path Preliminaries 117
7.3 The Recursive Algorithm 120
7.3.1 ℓ_1-Embedding from Approximate SSSP Potential 123
7.3.2 Sparsification and Recursion to Smaller Instances 124
7.4 ℓ_1-Oblivious Routing and Sequential Transshipment 125
7.4.1 Improved ℓ_1-Oblivious Routing: Our Techniques 126
7.4.2 Sherman's Framework 128
7.4.3 Polynomial Aspect Ratio 130
7.4.4 Reduction to ℓ_1 Metric 130
7.4.5 Oblivious Routing on ℓ_1 Metric 131
7.4.6 Parallel Transshipment 143
7.5 Vertex Sparsification and Recursion 143
7.5.1 Case $S = \{s\}$ of Lemma 7.3.15 144
7.5.2 Extending to Contracted Paths 145
7.5.3 Extending to Forest Components 148
7.5.4 Generalizing to S-SSSP 149
7.6 Ultra-spanner Algorithm 152
7.7 Sherman's Framework via Multiplicative Weights 155
7.8 Transshipment to Expected SSSP: Sequential 161
7.8.1 Parallelizing the Expected SSSP Algorithm 175
7.9 Sampling a Primal Tree 183
7.10 Omitted Proofs 187
7.10.1 Proof of Lemma 7.3.5 187
7.10.2 Proof of Lemma 7.3.12 188
7.10.3 Proof of Lemma 7.5.10 190
7.11 Conclusion 190

8 Deterministic Expander Decomposition **193**
8.1 Background 193
8.1.1 Our Techniques: Unweighted 194
8.1.2 Chapter Organization 199
8.2 Additional Preliminaries 200
8.2.1 Explicit Construction of Expanders 200
8.2.2 The Cut-Matching Game 200
8.2.3 Expander Pruning 201
8.2.4 Embeddings of Graphs and Expansion 201
8.2.5 Embeddings with Fake Edges and Expansion 202
8.2.6 j-trees 203

8.3 Route or Cut: Algorithm for the Matching Player 204
8.4 Deterministic Cut-Matching Game: Proof of Theorem 8.1.3 212
8.4.1 Base Case: $q = 1$ 212
8.4.2 Step: $q > 1$ 215
8.5 A Slower Algorithm for BalCutPrune 224
8.5.1 Extension of Theorem 8.1.3 to Smaller Sparsity 225
8.5.2 Degree Reduction 226
8.5.3 Completing the Proof of Theorem 8.5.1 228
8.6 Unweighted Expander Decomposition 232
8.6.1 Spectral Sparsification 233
8.7 Weighted Expander Decomposition with Custom Demands 237
8.7.1 Our Techniques 238
8.7.2 The WeightedBalCutPruneAlgorithm 240
8.7.3 Completing the Proof of Theorem 8.7.5 and Theorem 8.7.3 248
8.8 Weighted Expander Decomposition, Boundary-Linked 252
8.9 Conclusion 255

Bibliography **257**

Chapter 1

Introduction

Algorithms is a broad, rich, and fast-growing field. For the latter half of last century, many branches of algorithms have emerged and grown in popularity, and many different techniques have been invented to solve the central problems in each area. Some of these techniques, such as the *push-relabel* algorithm for maximum flow, are specially designed to solve a single problem. Other techniques, such as the multiplicative weights update method, are more general and applicable to a wide range of problems. And others, such as dynamic programming, divide and conquer, and linear programming relaxation and rounding, are so fundamental that they have not only pervaded every branch of algorithms, but have ultimately reshaped the way we approach algorithm design.

This thesis is devoted to studying two more modern algorithmic techniques, namely *preconditioning* and *locality*, which were pioneered by Spielman and Teng [100] in their ground-breaking work on Laplacian system solvers and have seen countless new applications in the past decade. In this thesis, we successfully apply preconditioning and locality to resolve fundamental open problems from a wide array of algorithmic subfields, from fast, sequential algorithms to deterministic algorithms to parallel algorithms, thereby demonstrating the power and versatility of the two techniques. Taking one step further, we make our case that preconditioning and locality are more than just powerful tools with countless applications: they are new, fundamental ways of thinking about algorithms that have the potential to revolutionize algorithm design just like dynamic programming and divide and conquer had done in the past.

In this introductory section, we first introduce the two techniques, preconditioning and locality, and provide the relevant history and background. Along the way, we illustrate how both techniques can be applied to the classic *minimum cut* problem on a graph, deriving surprisingly simple and fast algorithms for various settings of the problem.

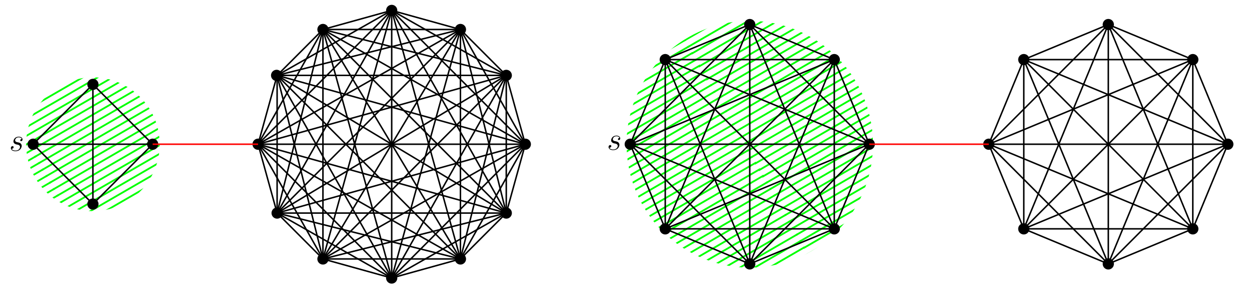

Figure 1.1: Both graphs above are composed of two cliques connected by a single (red) edge, the mincut of the graph. The green regions of each graph mark the area a local algorithm needs to explore, starting from the seed vertex s, before it detects the red edge and certifies it as a cut. The instance on the left is more amenable to a local algorithm than the one on the right.

1.1 Locality: Unbalanced vs. Balanced

In the modern, digital era, data sets have become so large that many algorithms cannot afford to even read in the whole input. In other words, even *linear*-time algorithms are often too slow in practice. This dilemma motivates the concept of *locality* in algorithm design: a *local* algorithm is one that only reads in data "local" to a *seed* location in the input. When the input is a massive graph, a local algorithm may only explore a small neighborhood around the seed vertex before outputting the solution. This is the approach taken by many local graph algorithms with theoretical guarantees, most notably the PageRank Nibble algorithm for computing a cut of small conductance around a seed vertex of a graph [8, 100].

Of course, for such an algorithm to be possible, the solution to the problem must also be local to the seed vertex. For illustration, consider the task of finding the *(global) minimum cut* of the graph, defined as the smallest set of edges whose deletion disconnects the graph. Consider the two graphs in Figure 1.1: both are comprised of two cliques attached by a single edge (marked red), and our task is to locate the red edge, which is a mincut of size 1. From the seed vertex s, a local algorithm can explore the clique containing s (highlighted in green) and then discover that the only edge neighboring that clique is the single red edge. This is enough information to certify that the red edge indeed forms a cut in the graph, at which point the algorithm can stop and output that cut. If the clique containing s is much smaller than the entire graph (Figure 1.1, left), then this algorithm avoids reading in most of the graph, which saves a lot of computation time. On the other hand, if the clique containing s occupies a large fraction of the entire graph (Figure 1.1, right), then the local algorithm still has to look at most of the graph. In other words, not every instance is amenable to a local algorithm: the instance on the left of Figure 1.1 is more "local" than the one on the right.

In this thesis, we advance beyond the concept of local algorithms and study locality as a fundamental *principle* in algorithm design. What is so special about locality that enables us to obtain faster algorithms? More precisely, suppose we have a problem instance where

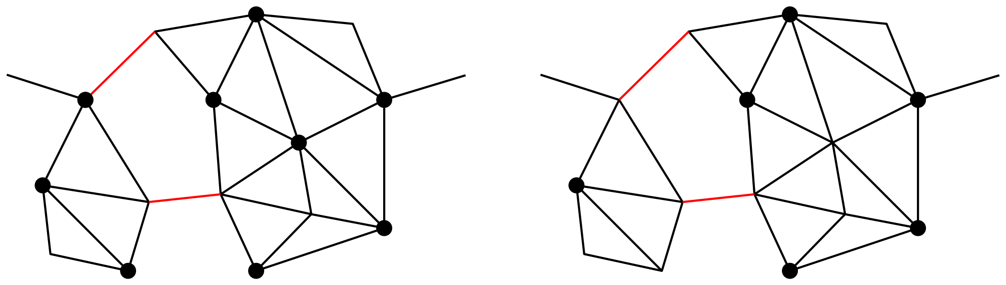

Figure 1.2: Instances of Steiner mincut where the terminal vertices are marked by dots. The instance to the right satisfies the locality assumption: the Steiner mincut (marked in red) separates exactly one terminal from the rest. By sampling terminals from the instance on the left at the correct sampling probability, we can create a new instance that satisfies the locality assumption (like the one on the right).

the target solution is local to a particular location of the input (possibly unknown to us). That is, we have a problem instance like the one to the left of Figure 1.1, not the one to the right. How can we solve the instance faster by exploiting this additional locality assumption, and what new techniques can we develop along the way? Lastly, since our goal is to design algorithms that work on all instances, not just local ones, we also need to remove the locality assumption eventually.

Application: Steiner mincut. We now highlight our locality-based approach to the mincut problem, which illustrates many of our techniques in a remarkably simple setting. We actually consider a more general problem called the global *Steiner* mincut, where we are given both an undirected graph and a subset of vertices called the terminals, and we want to find the minimum-weight set of edges whose removal disconnects at least two terminals from each other (see Figure 1.2). The (global) mincut is simply the Steiner mincut when every vertex is a terminal.

To approach this problem from a locality perspective, we first specify what it means to be local in our problem setting. We define a Steiner mincut to be local if it separates *exactly one* terminal from the rest, or in other words, the cut is “local” to that particular terminal (which is unknown to us). In Figure 1.2, the instance on the left does not have a local Steiner mincut, but the graph on the right does.

Given this locality assumption, we only need to look at cuts that separate one terminal from the others. That is, the task reduces to finding, for each terminal, the minimum number of edges that separate that terminal from the rest. We call this problem the *minimum isolating cuts* and design a fast and simple algorithm for it which is covered in Chapter 2. This solves the Steiner mincut problem under this locality assumption.

Finally, removing the locality assumption turns out to be miraculously easy: we simply *sample* a subset of terminals and declare the sample as the new set of terminals. By trying enough times at various sampling probabilities, we can ensure with high probability that for some sample, the Steiner mincut has exactly one sampled terminal on one side (see Figure 1.2, right). We can therefore solve Steiner mincut with the additional locality assumption on this sampled terminal set and obtain the Steiner mincut on the original instance.

This simple algorithm is covered entirely in the short Chapter 2 and Section 3.2, and yet it was the first nontrivial Steiner mincut algorithm on weighted graphs. Looking back, we believe that the key insight that eluded researchers in the past was viewing the problem from a locality perspective. Once the locality assumption is established, the remaining pieces fall together almost seamlessly.

1.1.1 Minimum Isolating Cuts and Applications

The aforementioned minimum isolating cuts problem turns out to play a central role in many of our locality-based algorithms. We recall the problem definition: the input is a graph with a subset of vertices as terminals, and the goal is to compute, for each terminal, the minimum cut that separates that terminal from the rest of the terminals. In other words, we want to compute a mincut "local" to each terminal.

Clearly, minimum isolating cuts can be trivially solved using $|R|$ calls to (s,t)-mincut. What is surprising is that we can do much better: we design a simple algorithm requiring only $O(\log|R|)$ calls to (s,t)-mincut. In other words, minimum isolating cuts and (s,t)-mincut have the same time complexity up to this logarithmic factor. We call this result the *isolating cuts lemma*, stated and proved in Chapter 2.

Global Steiner connectivity (Chapter 3). As mentioned before, one immediate consequence of the isolating cuts lemma is a randomized algorithm for global Steiner mincut problem in roughly (s,t)-mincut time, already the fastest known for general, weighted graphs. The simple algorithm and analysis is presented in Chapter 3.

Theorem: Randomized Steiner mincut (see Theorem 3.2.1)

There is a randomized Steiner mincut algorithm that runs in $O(\log^3 n)$ many calls to (s,t)-mincut.

We then derandomize our algorithm, which requires technical tools from derandomization as well as expander decompositions. Naturally, the running time also becomes larger, although for the current running time of (s,t)-mincut, the additional overhead is only polylogarithmic. Even for the deterministic global mincut problem, this was the first improvement over the $\tilde{O}(mn)$ time algorithm of Hao and Orlin [48]. We present the deterministic Steiner mincut algorithm in Section 3.3.

Theorem: Deterministic Steiner mincut (see Theorem 3.3.1)

For any constant $\epsilon > 0$, there is a randomized Steiner mincut algorithm that runs in polylog(n) many calls to (s,t)-mincut, plus $O(m^{1+\epsilon})$ additional time.

Gomory-Hu tree (Chapter 4). In Chapter 4, we use the isolating cuts lemma to develop locality-based algorithms that compute the Gomory-Hu tree, a classic data structure that encodes all pairwise (s,t)-mincuts of a graph. In particular, given the Gomory-Hu tree, we can answer any (s,t)-mincut queries in constant time per query.

Gomory and Hu [44] showed that a Gomory-Hu tree can be computed using $n-1$ many (s,t)-mincut computations, and for general, weighted graphs, this bound has yet to be improved sixty years later. While we were unable to break this barrier, we managed to reduce the Gomory-Hu tree problem to a seemingly much simpler problem, which we call *single-source mincut verification*: given a source vertex s and (s,t)-mincut overestimates $\tilde{\lambda}(s,t)$ for all other vertices t, determine which estimates $\lambda(s,t)$ equal the true (s,t)-mincut values.

Theorem: Gomory-Hu tree from single-source mincut verification (see Theorem 4.2.3)

There is a randomized Gomory-Hu tree algorithm that makes calls to single-source mincut verification on graphs with a total of $\tilde{O}(n)$ vertices and $\tilde{O}(m)$ edges, and runs for max-flow time outside of these calls.

Note that this problem is even simpler than single-source mincut, where we are given the source vertex s and need to compute the (s,v)-mincut for each $v \in V \setminus s$. Unfortunately, we do not know how to solve even the verification problem faster than computing a separate (s,v)-mincut for each $v \in V \setminus s$. Nevertheless, as a simpler, seemingly more tractable problem, single-source mincut verification may prove the key to obtaining faster Gomory-Hu tree algorithms in the future.

Approximate Gomory-Hu tree (Chapter 4). If we relax the problem to computing an *approximate* Gomory-Hu tree, then we can indeed obtain faster algorithms. Our main result in Section 4.7 is an algorithm for approximate Gomory-Hu tree that makes polylogarithmic calls to exact max-flow. This algorithm essentially follows from the fact that the reduction to single-source mincut verification is robust to approximations, and that approximate single-source mincut *can* be computed in faster time. To solve approximate single-source mincut, we introduce a new problem which we name the *cut threshold* problem: given a source vertex and a parameter λ called the cut threshold, find all vertices whose pairwise mincut with the source is at most λ.

Using the isolating cuts lemma, we show that the cut threshold problem can be solved in roughly max-flow time. We then reduce approximate single-source mincut to the cut threshold problem by trying geometrically increasing values of λ. Putting everything together, we obtain an approximate single-source mincut algorithm in roughly max-flow time, and by the aforementioned reduction, an approximate Gomory-Hu tree as well.

Theorem: Approximate Gomory-Hu tree (see Theorem 4.2.5)

There is a randomized $(1+\epsilon)$-approximate Gomory-Hu tree algorithm that makes calls to max-flow on graphs with a total of $\tilde{O}(n)$ vertices and $\tilde{O}(m)$ edges.

Directed global mincut (Chapter 5) Last but not least, we study the global mincut problem in directed graphs in Chapter 5 and reduce the problem to $O(\sqrt{n})$ many max-flow calls. Using the current best max-flow algorithm [75], our running time becomes $\tilde{O}(m\sqrt{n}+n^2)$, improving upon the $\tilde{O}(mn)$ bound of Hao and Orlin [48].

Theorem: Directed mincut (see Theorem 5.3.1)

There is a directed global mincut algorithm that makes $\tilde{O}(\sqrt{n})$ max-flow calls.

This is the only result of the locality section that does not rely on the isolating cuts lemma as a subroutine. Nevertheless, we approach the problem from a locality perspective, dividing the problem into an unbalanced and a balanced case. For the unbalanced case, where the optimal cut has a small number of vertices on one side, we use a locality-based approach similar to the one for deterministic mincut (Chapter 6). For the balanced case, we simply sample vertices s, t at random and compute an (s,t)-mincut, which succeeds with large enough probability that we can repeat the procedure a small number of times to succeed w.h.p.

1.2 Preconditioning: Worst Case vs. Average Case

Traditionally, algorithms are studied in the *worst-case* setting: for an algorithm to be deemed "fast", it must run quickly on *all* instances of the problem. The appeal of worst-case analysis is its robustness and its emphasis on concrete, universal statements: algorithms must work well for *all* inputs. However, designing optimal algorithms for the worst case is difficult, and it is often easier to consider average-case analysis, where algorithms are designed and analyzed for *well-behaved* instances, rather than for worst-case instances. These algorithms are often faster and cleaner, but they lack the strong universal guarantees of worst-case algorithms.

Preconditioning is a technique that strives for the best of both worlds: the optimality and simplicity of average-case analysis and the robustness of the worst-case setting. This is achieved by transforming, or *preconditioning*, any input instance to behave like an average-case instance. In this way, we can focus our attention on well-behaved, non-pathological

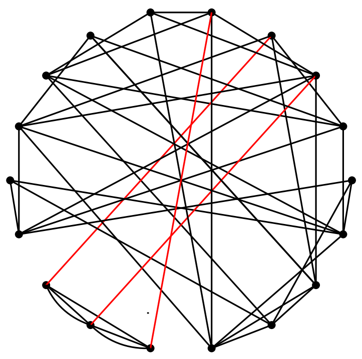

Figure 1.3: An expander graph with the global mincut marked in red. The global mincut is "unbalanced" since one of its sides only has 3 vertices.

instances, and then translate the results back to derive worst-case bounds.

Historically, preconditioning was first developed in the field of numerical linear algebra to solve linear systems of the form $Ax = b$. While the matrix A may be difficult to solve in the worst case, one can, for example, multiply both sides by a matrix M to arrive at the equivalent $(MA)x = Mb$. In this context, preconditioning is the art of choosing the matrix M so that the new, preconditioned matrix MA is well-behaved. One can then apply iterative methods which converge in few iterations on a well-conditioned matrix. The first paper to utilize the term *preconditioning* is due to Evans [35], though the concept has been employed to solve linear systems by hand over a century ago [92].

In the context of graph algorithms, preconditioning can be applied in various ways depending on the problem being solved. For distance-related problems, Awerbuch et al. [12] introduced the concept of *low-diameter network decompositions* which are now standard in shortest path and related algorithms. This technique decomposes a general graph into subgraphs of small diameter, which often admit faster and simpler distance-based algorithms. For graph cut and spectral problems, Spielman and Teng popularized the technique of *expander decomposition* in their seminal work on solving linear systems on graphs [100]. Here, general graphs are decomposed into *expanders*, graphs that exhibit nice cut and spectral properties. We study both low-diameter and expander decompositions in this thesis.

Application: deterministic mincut. Once again, we illustrate how we can apply the preconditioning technique to the mincut problem.

Informally, the preconditioning technique reduces general graph instances to the case when the input graph is an expander. The key property that we exploit is that on an expander, the global mincut is *unbalanced* in the locality sense: one of its sides has very few vertices (see Figure 1.3). This shows how the concepts of preconditioning and locality often go hand in hand.

How is this guarantee useful for us? While the locality-based Steiner mincut algorithm works on any graph, it crucially relies on a random sampling procedure, which makes the algorithm inherently randomized. Fortunately, in an expander where the global mincut is unbalanced, the random sampling procedure can be efficiently *derandomized*. The derandomization technique does not work well on general graphs, however, which is where preconditioning comes in handy: we precondition the graph to behave like an expander by computing an *expander decomposition* of the graph. The final result is a deterministic Steiner mincut algorithm, discussed in Section 3.3.

1.2.1 Graph Cut Problems

In this thesis, all problems studied include graphs as part of their input. For graph cut problems, such as global mincut, the average-case instances that we study are the *expander graphs*. This is a natural class of graphs to study in average-case analysis since it includes the *random graphs*, e.g., those drawn from the Erdös-Rényi $G(n, p)$ model where each (undirected) edge is independently sampled with probability p.

By assuming that the input graph is an expander, we can appeal to the rich theory of expanders, including the connection to spectral graph theory established by Cheeger's inequality. As for preconditioning the graph, or transforming it into the average case setting, our main strategy is to compute an *expander decomposition* of the input graph: we delete a small fraction of edges so that each connected component of the remaining graph is an expander. This simple concept, popularized by Spielman and Teng in their seminal paper on solving Laplacian systems, has proven invaluable in recent breakthroughs in fast graph algorithms in many areas, from sequential to dynamic to distributed algorithms. Following an expander decomposition, a common strategy is to first solve the problem separately on each expander by appealing to average-case analysis. Then, to handle the edges deleted by the decomposition, we apply recursion on an instance a constant factor smaller in order to keep the overall running time small. While most preconditioning-based approaches follow the same general outline, tailoring the method to each individual problem is always the key challenge and, ultimately, lies at the core of the art of preconditioning.

Deterministic preconditioning (Chapter 8). Prior to our work, the biggest drawback to expander decomposition-based algorithms was that the only near-linear time algorithms to compute an expander decomposition were randomized. In joint work with Chuzhoy, Gao, Nanongkai, Peng, and Saranurak [27], we develop the first almost-linear time, *deterministic* algorithm for expander decomposition, thus opening the door to deterministic, preconditioning-based algorithms. Due to the technical complexity of the algorithm, we defer it to Chapter 8 at the end of the thesis.

Deterministic mincut (Chapter 6). We next discuss a new application of deterministic expander decomposition, namely to the *global mincut* problem: determine the smallest-weight set of edges whose removal disconnects the graph.

A classic result of Karger [55] established a near-linear time randomized algorithm for global mincut, and Karger famously posed as an open question whether a fast, *deterministic* algorithm exists. For almost twenty years, no progress had been made towards even partially answering this question, until the breakthrough result of Kawarabayashi and Thorup [57] who achieved a deterministic, near-linear time algorithm on *simple*, unweighted graphs. Their key conceptual contribution is a simple but meaningful connection between mincuts and expanders: on an expander, the global mincut must be *unbalanced* in the locality sense. Their work serves as evidence that the global mincut problem should be much easier on an expander, which suggests a preconditioning-based line of attack.

In recent work [69], we manage to complete the preconditioning approach for the deterministic mincut problem, resolving Karger's open problem from the 1990s.

Theorem: Deterministic mincut (see Theorem 6.2.1)

There is a deterministic mincut algorithm that runs in $m^{1+o(1)}$ time.

Our strategy is to de-randomize the single randomized component in Karger's mincut algorithm, namely the construction of the graph *sparsifier* by random sampling. We adopt a preconditioning-based approach, first solving the case when the input graph is an expander, and then applying expander decomposition to generalize to all graphs. To sparsify an expander, we exploit the locality of the target mincut as mentioned before (see Figure 1.3). In particular, it suffices to preserve only the unbalanced cuts in the sparsification procedure, which makes it much easier to derandomize.

For the general case, our strategy is still to preserve a subset of "unbalanced" cuts, but this time, the notion of unbalanced is defined with respect to a recursive expander decomposition hierarchy of the graph. We defer the details to our full presentation of the algorithm in Chapter 6.

1.2.2 Graph Distance Problems

For graph problems involving distances, such as the single-source shortest path problem, the property we exploit from average-case instances is that they have small *aspect ratio*, defined as the ratio of the maximum to minimum distances between distinct vertices of the graph. Once again, this assumption is consistent with the theory of random graphs in average-case analysis: for a random graph sampled from the Erdös-Rényi $G(n, p)$ model, where all edges sampled have unit weight, the aspect ratio is only $O(\log n)$.

Parallel shortest path (Chapter 7). We apply distance-based preconditioning to the single source shortest path problem in the parallel (PRAM) setting. The parallel shortest path problem is notoriously difficult, especially in the exact setting, since classic shortest path algorithms such as Dijkstra's are inherently sequential. This barrier encouraged researchers to study the *approximate* single-source shortest path problem instead, for which near-optimal parallel algorithms are now known. A classical result of Cohen [28] developed an algorithm based on the concept of *hopsets* that runs in $O(m^{1+\epsilon_0})$ work and polylogarithmic time for any constant $\epsilon_0 > 0$, which is optimal up to the m^{ϵ_0} factor. The natural follow-up question is whether the work can be improved to $m \operatorname{polylog}(n)$, but this question has resisted two decades of attempts.

In our paper [67], we tackle this problem from a continuous perspective, using the preconditioning-based method of Sherman [96, 97]. We reduce the SSSP problem to the more continuous *minimum transshipment* problem, also known as uncapacitated minimum cost flow, and provide $(1+\epsilon)$-approximate algorithms for both problems in $m \operatorname{polylog}(n)$ work and polylogarithmic time. This improves upon Cohen's result and achieves the targeted optimality.

Theorem: Parallel SSSP and transshipment (see Theorems 7.1.1 and 7.1.2)

There are parallel $(1+\epsilon)$-approximate SSSP and transshipment algorithms that run in $\tilde{O}(m)$ work and $\operatorname{polylog}(n)$ time.

Sherman's preconditioning-based method is based on his key insight that low-diameter graphs admit a simple transshipment algorithm. To precondition a general graph into low-diameter graphs, he computes low-diameter decompositions of the graph at varying diameter scales. For technical reasons, the low-diameter decomposition is performed on an *embedding* of the graph into high-dimensional (Euclidean) space, where distances between vertices in the graph are approximated by distances between their corresponding points in space. We leave the details to the full presentation in Chapter 7.

1.3 Preliminaries

In this thesis, all graphs are either undirected or directed, and either unweighted or weighted on the edges. All weighted graphs have positive weights, and unweighted graphs are treated as weighted graphs with weight 1 on all edges. We allow multiple parallel edges, even on weighted graphs, but no self-loops unless explicitly stated otherwise.

We first start with standard graph-theoretic notation. For an undirected graph $G = (V, E)$ and two vertex subsets $A, B \subseteq V$, we denote by $E(A, B)$ the set of all edges with one endpoint in A and another in B. If G is directed, then $E_G(A, B)$ denotes the set of (directed) edges from a vertex in A to a vertex in B. For an edge $e \in E$, let $w(e)$ be the weight of that edge, and for vertices $u, v \in V$, let $w(u, v)$ be the *sum* of the weights $w(e)$ of all (parallel) edges e between u and v. For a vertex subset $S \subseteq V$, define $\partial S := E(S, V \setminus S)$. For a subset $F \subseteq E$ of edges, denote its total weight by $w(F)$. Define the (weighted) degree of a vertex $v \in V$ as $w(\partial(\{v\}))$, and for a subset $S \subseteq V$, define its volume $\mathbf{vol}(S) := \sum_{v \in S} \deg(v)$. For a subset $S \subseteq V$, define $N(S) \subseteq V$ as the set of vertices $v \in V \setminus S$ with a neighbor in S, and define $N[S] = S \cup N(S)$. When the graph G is ambiguous, we may add a subscript of G

in our notation, such as $E_G(A, B)$. When a set in question is a singleton vertex v, we may write v instead of $\{v\}$, such as in $E(v, B)$. Finally, for any graph H, we use $V(H)$ to denote the set of vertices of H, and $E(H)$ to denote the set of edges of H.

For an undirected graph $G = (V, E)$, we define an (edge) *cut* to be either (1) the set of vertices S on one side of the cut, (2) the corresponding bipartition $(S, V \setminus S)$ of vertices, or (3) the edges ∂S in the cut. We alternate between the three definitions depending on which one is most convenient for the occasion.

1.4 Bibliographic Notes

Most of this thesis is based on previously published work.

1. Chapters 2 and 3 are based on the publication "Deterministic Min-cut in Poly-logarithmic Max-flows" [70].
2. Chapter 4 is based on the publication "Approximate Gomory-Hu Tree Is Faster than $n - 1$ Max-flows" [71].
3. Chapter 5 is based on the publication "Minimum Cuts in Directed Graphs via $\sqrt{n}$ Max-Flows" [18].
4. Chapter 6 is based on the publication "Deterministic Mincut in Almost-Linear Time" [69].
5. Chapter 7 is based on the publication "Faster Parallel Algorithm for Approximate Shortest Path" [67].
6. Chapter 8 is based on the publication "A Deterministic Algorithm for Balanced Cut with Applications to Dynamic Connectivity, Flows, and Beyond" [27].

Omitted Work. This thesis does not include a number of other results that the author has published during his PhD. The most notable such results, in the author's opinion, are listed below.

1. The Karger-Stein Algorithm Is Optimal for k-cut [46].
2. A Quasipolynomial $(2 + \epsilon)$-Approximation for Planar Sparsest Cut [30].
3. The Connectivity Threshold for Dense Graphs [47].
4. Tight FPT Approximations for k-Median and k-Means [29].

Part I

Locality

Chapter 2

Minimum Isolating Cuts

In this chapter, we study the *minimum isolating cuts* problem introduced in [70]: given a list of terminals, compute, for each terminal, the mincut separating it from the other terminals. We present the simple algorithm from [70] that solves this problem in a logarithmic number of (s,t)-mincut calls.

In just one year after its introduction, the minimum isolating cuts problem has already seen numerous applications to graph cut algorithms. In Chapters 3 and 4 of the thesis, we use the minimum isolating cuts algorithm as a core subroutine in our Steiner mincut and Gomory-Hu tree algorithms. The minimum isolating cuts has also appeared in problems ranging from vertex connectivity [73] to submodular function minimization [23, 84], which are outside the scope of this thesis.

Given the simplicity of the algorithm and its analysis, as well as its importance in recent developments, it is perhaps miraculous that the minimum isolating cuts problem and algorithm had remained undiscovered for so long. In hindsight, we believe the biggest hurdle to its discovery was not the technical algorithm itself, but the conceptual realization that the minimum isolating cuts problem could be so useful in the first place, especially in locality-based algorithms. In fact, we attribute our discovery of the problem entirely to our locality perspective: as we will see in Chapter 3, it is exactly the problem to study when solving Steiner mincut with a locality assumption.

2.1 Background

The minimum isolating cuts problem was first introduced to solve the deterministic mincut problem [70]. Since its inception, the isolating cuts lemma has been generalized to the wider setting of symmetric, submodular functions [23, 84], leading to new developments on submodular function minimization and hypergraph mincut.

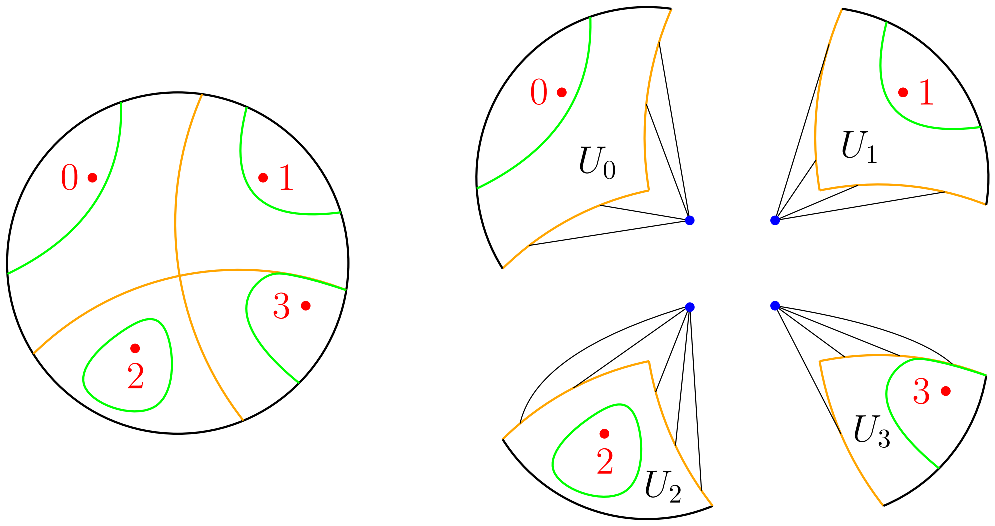

Figure 2.1: The minimum isolating cuts algorithm for $|R| = 4$. The orange marks the "upper boundary" of each green isolating cut. They are formed by the min-cut separating $\{0, 1\}$ and $\{2, 3\}$ and the min-cut separating $\{0, 2\}$ and $\{1, 3\}$.

2.2 The Isolating Cuts Algorithm

We first formally define the minimum isolating cuts problem.

Definition 2.2.1: Minimum isolating cuts

Consider a weighted, undirected graph $G = (V, E)$ and a subset of vertices $R \subseteq V$ where $|R| \geq 2$. The *minimum isolating cuts* for R is a collection of sets $\{S_v : v \in R\}$ such that for each vertex $v \in R$, the set S_v satisfies $S_v \cap R = \{v\}$ and $w(\partial S'_v)$ is a $(v, R \setminus v)$-mincut.

We now state the isolating cuts lemma, whose proof occupies the rest of this section.

Lemma 2.2.2: Isolating cuts lemma

Fix a subset $R \subseteq V$ of terminals, where $|R| \geq 2$. There is an algorithm that computes the minimum isolating cuts for R using $\lceil \lg |R| \rceil + 1$ calls to (s, t)-mincut on weighted graphs of $O(n)$ vertices and $O(m)$ edges, and takes $\tilde{O}(m)$ deterministic time outside of the mincut calls. If the original graph G is unweighted, then the inputs to the mincut calls are also unweighted.

Our main idea is to first compute, for each terminal, an "upper boundary" to the location of the mincut separating that terminal from the rest (see Figure 2.1). More precisely, for each terminal $v \in R$, we want to compute a set U_v of vertices that contains S_v as defined in Definition 2.2.1. If we can do so, then it suffices to compute an (v, t)-mincut on the graph G

with $V \setminus U_v$ contracted to a single vertex t, which will return ∂S_v or some other $(v, R \setminus v)$-mincut. To make this mincut computation fast, we would like U_v to be small, ideally not much larger than S_v. We are not able to prove such a strong local guarantee, but we can ensure that the sets U_v are *disjoint* among all $v \in R$. In other words, some terminal $v \in R$ might have small S_v and linear-sized U_v, but this cannot happen for too many terminals, since $\sum_{v\in R} |U_v| \le n$ must hold.

Our procedure to compute the sets U_v is as follows. We first compute $\lceil \lg |R| \rceil$ many bipartitions of R such that any two terminals are separated in at least one bipartition. For each bipartition (A, B) of R, we compute a (A, B)-mincut, and then for each terminal $v \in R$, we set U_v as the common intersection of the sides containing v of the $\lceil \lg |R| \rceil$ many computed mincuts. We show by a simple submodularity argument that the side containing v of each of the $\lceil \lg |R| \rceil$ mincuts must contain S_v (if we assume S_v to be minimal in a sense), and thus, their common intersection U_v also contains S_v.

For the rest of this section, we formalize the above intuition and prove Lemma 2.2.2.

Proof of the isolating cuts lemma (Lemma 2.2.2). Order the vertices in R arbitrarily from 1 to $|R|$, and let the *label* of each $v \in R$ be its position in the ordering, a number from 1 to $|R|$ that is denoted by a unique binary string of length $\lceil \lg |R| \rceil$. Let us repeat the following procedure for each $i = 1, 2, \ldots, \lceil \lg |R| \rceil$. For each vertex v, color it *red* if the i'th bit of its label is 0, and *blue* if the i'th bit of its label is 1. Then, compute a min-cut $C_i \subseteq E$ in G between the red vertices and the blue vertices (for iteration i).

First, we show that $G \setminus \bigcup_i C_i$ partitions the set of vertices into connected components each of which contain at most one vertex of R. Let U_v be the connected component in $G \setminus \bigcup_i C_i$ containing $v \in R$. Then:

Claim 2.2.3

$U_v \cap R = \{v\}$ for all $v \in R$.

Proof. By definition, $v \in U_v \cap R$. Suppose for contradiction that $U_v \cap R$ contains another vertex $u \ne v$. Since the binary strings assigned to u and v are distinct, they differ in their j'th bit for some j. Then, the cut C_j must separate u and v, i.e., removing the edges in C_j leaves u and v in separate components, which is a contradiction. □

Now, for each vertex $v \in R$, let λ_v be the minimum value of $w(\partial S)$ over all $S \subseteq V$ satisfying $S \cap R = \{v\}$, and let $S_v^* \subseteq V$ be an inclusion-wise minimal set satisfying $S_v^* \cap R = \{v\}$ and $w(\partial S_v^*) = \lambda_v$. Then, we claim that the cut S_v^* does not *cross* the cut U_v, i.e.:

Claim 2.2.4

$U_v \supseteq S_v^*$ for all $v \in R$.

Proof. Fix a vertex $v \in V$ and an iteration i. Let the side of the cut C_i containing v be $T_v^i \subseteq V$; we claim that $S_v^* \subseteq T_v^i$. Suppose for contradiction that $S_v^* \setminus T_v^i \ne \emptyset$. Note that

$(S_v^* \cap T_v^i) \cap R = \{v\}$, which implies that:

$$w(\partial(S_v^* \cap T_v^i)) \geq \lambda_v = w(\partial S_v^*).$$

Indeed, by our choice of S_v^* to be inclusion-wise minimal, we can claim the strict inequality:

$$w(\partial(S_v^* \cap T_v^i)) > \lambda_v = w(\partial S_v^*).$$

But, by submodularity of cuts, we have:

$$w(\partial(S_v^* \cup T_v^i)) + w(\partial(S_v^* \cap T_v^i)) \leq w(\partial S_v^*) + w(\partial T_v^i).$$

Therefore, we get:

$$w(\partial(S_v^* \cup T_v^i)) < w(\partial T_v^i).$$

But $(S_v^* \cup T_v^i) \cap R = T_v^i \cap R$ since $(S_v^* \setminus T_v^i) \cap R = \emptyset$. In particular, the cut $\partial(S_v^* \cup T_v^i)$ also separates red vertices from blue vertices in the ith iteration. This contradicts the choice of $\partial T_v^i = C_i$ as the min-cut separating red vertices from blue vertices in the ith iteration.

Therefore, over all iterations i, none of the edges in the induced subgraph $G[S_v^*]$ are present in C_i. Note that $G[S_v^*]$ is a connected subgraph; therefore, it is a subgraph of the connected component U_v of $G \setminus \bigcup_i C_i$ containing v. □

It remains to compute the desired set S_v given the property that $U_v \supseteq S_v$. Starting from G, contract $V \setminus U_v$ into a single vertex t; we want to compute the min v–t cut in the contracted graph G_v, which corresponds to a set S_v satisfying $S_v \cap R = \{v\}$ by Claim 2.2.3. Since $\partial_{G_v} S_v^*$ is a valid v–t cut in this graph by Claim 2.2.4, we have $w(\partial_{G_v} S_v) \leq w(\partial_{G_v} S_v^*) = w(\partial_G S_v^*) = \lambda_v$, as desired.

Note that each edge in E is either in exactly one graph G_v, or it is adjacent to t in exactly two graphs G_v. Therefore, the total number of edges over all graphs G_v is at most $2m$. We can compute the v–t min-cuts on all G_v in "parallel" through a single max-flow call on the disjoint union of all G_v. Note that if the original graph G is unweighted, then this max-flow instance is also unweighted. Finally, recovering the sets S_v and the values $w(\partial S_v)$ take time linear in the number of edges of G_v, which is $O(m)$ time over all $v \in R$.

This completes the proof of Lemma 2.2.2. □

2.3 Conclusion

In just one year since its discovery, the minimum isolating cuts lemma has seen many applications in graph cut problems from Gomory-Hu tree (Chapter 4) to vertex connectivity [73], as well as more abstract problems like bisubmodular function minimization [23, 84]. We anticipate many more future applications to come, and we expect the technique to reshape how we approach locality in graph algorithms in the same way expander decomposition did for preconditioning.

Chapter 3

Steiner Mincut

This chapter focuses on the (global) Steiner mincut problem: given an undirected graph and a subset T of vertices (called the *terminals*), the (global) *Steiner mincut* is the smallest-weight set of edges whose removal disconnects at least two vertices in T. The Steiner mincut is the natural generalization of both the global mincut and (s,t)-mincut problems. Nevertheless, prior to the results of this chapter, no nontrivial algorithm for Steiner cut was known in general.

In this chapter, we show that the isolating cuts lemma from Chapter 2 leads to a simple and elegant algorithm for Steiner mincut that uses polylogarithmic many calls to (s,t)-mincut. For weighted graphs and unweighted graphs of large enough Steiner connectivity, this is the fastest algorithm known. More importantly, since Steiner mincut also generalizes (s,t)-mincut, we conclude that both problems have the same time complexity up to polylogarithmic factors. We then derandomize our Steiner mincut algorithm using standard derandomization tools along with expander decompositions. At the time of publication, our algorithm was the fastest for even the special case of deterministic global mincut.

It is worth emphasizing again the simplicity of the randomized Steiner mincut algorithm. The only step in addition to the isolating cuts algorithm of Chapter 2 is an initial random sampling process. Namely, we sample terminals at varying sampling probabilities, and for each sample, we compute the minimum isolating cuts on the sampled terminals. A simple argument shows that with high probability, at least one of the isolating cuts computed over all samples is the minimum Steiner mincut.

It is rather remarkable that such a simple yet state-of-the-art Steiner mincut algorithm remained undiscovered for so long. Once again, we attribute our success to our locality perspective: as mentioned in the beginning of Chapter 2, viewing the Steiner mincut problem from a locality perspective is what led us to the minimum isolating cuts problem in the first place!

3.1 Background

For unweighted graphs, the fastest algorithm for Steiner mincut, due to Bhalgat et al. [16], runs in $\tilde{O}(m + nc^2)$ time where c is the size of the Steiner mincut. For weighted graphs, nothing nontrivial was known before our result.

The Steiner mincut problem is an important subroutine for fast Gomory-Hu tree algorithms. For example, a Gomory-Hu tree can be constructed using $n - 1$ calls to Steiner mincut instead of (s, t) max-flow. Bhalgat et al. [16] follow this approach to compute a Gomory-Hu tree, except they show that computation can be reused between the $n - 1$ calls of their specific Steiner mincut algorithm, speeding up their Gomory-Hu tree algorithm to $\tilde{O}(mn)$ time in total. We remark that their speed-up methods do not apply to our Steiner mincut algorithm, so we do not obtain a faster Gomory-Hu tree algorithm as a corollary.

3.2 Randomized Steiner Mincut

In this brief section, we show that the isolating cuts lemma (Lemma 2.2.2), along with a simple random sampling procedure, implies a randomized Steiner mincut algorithm that makes polylog(n) many max-flow calls.

Theorem 3.2.1: Randomized Steiner mincut

There is a randomized, Monte Carlo algorithm for Steiner mincut for weighted undirected graphs that makes $O(\log^3 n)$ calls to (s, t) max-flow on a weighted, undirected graph with $O(n)$ vertices and $O(m)$ edges. If the original graph G is unweighted, then the inputs to the max-flow calls are also unweighted.

Proof. The algorithm essentially calls Lemma 2.2.2 $O(\log^2 n)$ times; in each iteration, $R \subseteq T$ is a random set of vertices sampled at a particular scale.

For each positive integer $i \le \lg |T|$, repeat the following procedure $O(\log n)$ times: let $R \subseteq V$ be a random sample of 2^i vertices, and call Lemma 2.2.2 on the set R to obtain a cut S_v for each $v \in R$. Return the cut S with the minimum value of $w(\partial S_v)$ over all v and over all the iterations.

We claim that w.h.p., the returned cut S is a Steiner mincut. Let $S^* \subseteq V$ be the side of the Steiner mincut containing the smaller number of terminals. Observe that if, in any iteration, the sampled set R satisfies $|R \cap S^*| = 1$, then Lemma 2.2.2 will find the Steiner mincut. Consider the integer $i = \lfloor \lg(n/|S^*|) \rfloor$. Then, for each iteration where R is a random sample of size 2^i, we sample exactly one vertex in S^* with probability $\Omega(1)$. Since we sample at this scale $O(\log n)$ times, this occurs at least once w.h.p. □

3.3 Deterministic Steiner Mincut

In this section, we present our deterministic Steiner mincut algorithm.

Theorem 3.3.1: Deterministic Steiner mincut

Fix any constant $\epsilon > 0$. There is a deterministic algorithm for global Steiner mincut that makes $(\lg n)^{O(1/\epsilon^4)}$ calls to (s, t) max-flow on a weighted undirected graph with $O(n)$ vertices and $O(m)$ edges, and runs in $O(m^{1+\epsilon})$ time outside these max-flow calls. If the original graph G is unweighted, then the inputs to the max-flow calls are also unweighted.

Throughout the algorithm, we maintain a set $U \subseteq T$ of vertices that starts out as $U = T$ and shrinks over time. (Think of this set as the set R over which we call the isolating cuts lemma.) We distinguish between the cases when U is *k-unbalanced* or *k-balanced* for some $k = \text{polylog}(n)$, as defined below.

Definition 3.3.2: k-unbalanced, k-balanced

For any positive integer k, a subset $U \subseteq T$ is *k-unbalanced* if there exists a side $S \subseteq V$ of some Steiner mincut satisfying $1 \le |S \cap U| \le k$. More specifically, we say that U is *k-unbalanced with witness S*. The subset $U \subseteq T$ is *k-balanced* if there exists a Steiner mincut whose two sides S_1, S_2 satisfy $|S_i \cap U| \ge k$ for both $i = 1, 2$. More specifically, we say that U is *k-balanced with witness* (S_1, S_2).

We will only use this definition for subsets $U \subseteq T$ that span both sides of some Steiner mincut, i.e., $S \cap U \neq \emptyset$ and $(V \setminus S) \cap U \neq \emptyset$ for some Steiner mincut S. By definition, such a subset $U \subseteq T$ is either k-unbalanced or k-balanced (or possibly both, if there are multiple Steiner mincuts in the graph). If U is k-unbalanced with witness S for some $k = \text{polylog}(n)$, then the algorithm computes a family $\mathcal{F}$ of subsets of U of size $k^{O(1)}\text{polylog}(n) = \text{polylog}(n)$ such that some subset $R \in \mathcal{F}$ satisfies $|R \cap S| = 1$. The algorithm then executes the isolating cuts lemma (Lemma 2.2.2) on each subset in $\mathcal{F}$, guaranteeing that the target set R is processed and the Steiner mincut is found. Otherwise, U must be k-balanced with some witness (S_1, S_2). In this case, the algorithm computes a subset $U' \subseteq U$ such that $|U'| \le |U|/2$ and both $S_1 \cap U' \neq \emptyset$ and $S_2 \cap U' \neq \emptyset$. Of course, the algorithm does not know which case actually occurs, so it executes both branches. But the second branch can only happen $O(\log n)$ times before $|U| \le k$, at which point we can simply run (s, t) min-cut between all vertex pairs in U.

The algorithm is presented in Algorithm 1.

3.3.1 Unbalanced Case

In this section, we solve the case when U is k-unbalanced (line 4) for some fixed $k = \text{polylog}(n)$.

Algorithm 1 Deterministic Steiner mincut on $(G = (V, E))$

```
1: $U \leftarrow T$
2: $k \leftarrow C \log^C n$ for a sufficiently large constant $C = O(1/\epsilon^4)$
3: while $|U| \geq k$ do
4:     Run Lemma 3.3.3 on $U$    ▷ Handles case when $U$ is $k$-unbalanced (see Definition 3.3.2)
5:     Compute $U'$ from $U$ according to Lemma 3.3.7    ▷ Handles case when $U$ is $k$-balanced
6:     Update $U \leftarrow U'$    ▷ $|U|$ shrinks by at least factor 2
7: for each pair of distinct $s, t \in U$ do
8:     Compute min $(s, t)$ cut in $G$
9: return smallest cut seen in lines 4 and 8
```

Lemma 3.3.3: Unbalanced case

Consider a graph $G = (V, E)$, a parameter $k \geq 1$, and a k-unbalanced set $U \subseteq T$. Then, we can compute the Steiner mincut in $k^{O(1)}\text{polylog}(n)$ (s, t) max-flow computations plus $\tilde{O}(m)$ deterministic time.

Our goal is to de-randomize the simple random process of sampling each vertex independently with probability $1/k$. We compute a deterministic family of subsets $R \subseteq U$ such that for any subset S satisfying $|S \cap U| \leq k$ (in particular, for the Steiner mincut witnessing the fact that U is k-unbalanced), there exists a subset R in the family with $|R \cap S| = 1$. We call such a subset R an *isolator*.

Lemma 3.3.4: Deterministic isolator

For every n and $k < n$, there is a deterministic algorithm that constructs a family $\mathcal{F}$ of subsets of $[n]$ such that, for every non-empty subset $S \subseteq [n]$ of size at most k, there exists a set $T \in \mathcal{F}$ with $|S \cap T| = 1$. The family $\mathcal{F}$ has size $k^{O(1)} \log n$, every set in the family has at least two elements, and the algorithm takes $k^{O(1)} n \log n$ time.

Before we prove Lemma 3.3.4, we first show why it implies an algorithm for the unbalanced case as promised by Lemma 3.3.3.

Proof of Lemma 3.3.3. Let S be the Steiner mincut witnessing the fact that U is k-unbalanced. Apply Lemma 3.3.4 with parameters $n = |U|$ and k. Map the elements of $[n]$ onto U, obtaining a family $\mathcal{F}$ of subsets of U such that for any set $S' \subseteq U$ with $|S'| \leq k$, there exists a set $R \in \mathcal{F}$ with $|R| \geq 2$ and $|R \cap S'| = 1$. In particular, for the set $S' = S \cap U$, there exists $R \in \mathcal{F}$ with $1 = |R \cap S'| = |R \cap (S \cap U)| = |R \cap S|$. Invoke Lemma 2.2.2 on the set R to obtain, for each $v \in R$, a set S_v satisfying $S_v \cap R = \{v\}$ that minimizes $w(\partial S_v)$, along with the value $w(\partial S_v)$. Finally, output the set S_v with minimum value of $w(\partial S_v)$. To show that S_v is a Steiner mincut of graph G, it suffices to verify that S_v is a valid cut (that is, $\emptyset \subsetneq S_v \subsetneq V$), and that $w(\partial S_v) \leq w(\partial S)$.

Since $|R| \geq 2$, the set S_v satisfies $\emptyset \subsetneq S_v \subsetneq R$, so it is a cut of the graph G. Since $|R \cap S| = 1$, for the vertex $u \in U$ with $R \cap S = \{u\}$, the set S satisfies the constraints for S_u. In particular, $w(\partial S_u) \leq w(\partial S)$. We output the set S_v minimizing $w(\partial S_v)$, so $w(\partial S_v) \leq w(\partial S_u) \leq w(\partial S)$, as promised. □

The rest of this section focuses on proving Lemma 3.3.4. We first prove an easier variant, where we do not insist that every set in the family has at least two elements.

Lemma 3.3.5: Deterministic isolator, singletons allowed

For every n and k, there is a deterministic algorithm that constructs a family $\mathcal{F}$ of subsets of $[n]$ such that, for each subset $S \subseteq [n]$ of size at most k, there exists a set $T \in \mathcal{F}$ with $|S \cap T| = 1$. The family $\mathcal{F}$ has size $k^{O(1)} \log n$ and the algorithm takes $k^{O(1)} n \log n$ time.

To prove Lemma 3.3.5, we use the following de-randomization building block due to [7]. The theorem below is from [31], who state it in terms of (n, k, k^2)-*splitters* (which we will not define here for simplicity).

Theorem 3.3.6: Deterministic splitters (Theorem 5.16 from [31])

For any $n, k \geq 1$, one can construct a family of functions from $[n]$ to $[k^2]$ such that for every set $S \subseteq [n]$ of size k, there exists a function f in the family whose values $f(i)$ are distinct over all $i \in S$. The family has size $k^{O(1)} \log n$ and the algorithm takes time $k^{O(1)} n \log n$.

Proof of Lemma 3.3.5. Apply Theorem 3.3.6 to n and k, and for each function $f : [n] \to [k^2]$ in the constructed family, add the sets $f^{-1}(j)$ for all $j \in [k^2]$ to our family $\mathcal{F}$ of subsets of $[n]$. Fix any set $S \subseteq [n]$ of size k. For the function f guaranteed by Theorem 3.3.6 for this set S, we have $|f^{-1}(f(i)) \cap S| = 1$ for any $i \in S$. Therefore, setting $T = f(i)$ for any $i \in S$ suffices.

This only handles subsets $S \subseteq [n]$ of size *exactly* k, but we can repeat the above construction for each positive integer $k' \leq k$. The total size and running time go up by a factor of k, which is absorbed by the $k^{O(1)}$ factors. □

Finally, to prove Lemma 3.3.4, we add the condition that $\mathcal{F}$ cannot contain sets of size at most 1. Here, we will impose the additional constraint that $k < n$.

Proof of Lemma 3.3.4. The only difference in the output is that $\mathcal{F}$ must contain no sets of size at most 1. Apply Lemma 3.3.5 to n and k to obtain a family $\mathcal{F}_0$. Initialize a set $\mathcal{F}$ as $\mathcal{F}_0$ minus all subsets of size at most 1. For each singleton set $\{x\} \in \mathcal{F}_0$, choose k arbitrary elements in $[n] \setminus x$, and for each chosen element y, add the set $\{x, y\}$ to $\mathcal{F}$. The total size of $\mathcal{F}$ increases by at most a factor k. Now consider a subset $S \subseteq [n]$ of size at most k, and let T be a set in $\mathcal{F}_0$ with $|S \cap T| = 1$, as promised by Lemma 3.3.5. If $|T| > 1$, then $T \in \mathcal{F}$ as well. Otherwise, if $T = \{x\}$, then since $|S \setminus x| < k$ and we chose k elements $y \in [n] \setminus x$, there exists some chosen $y \notin S$ for which $\{x, y\}$ was added to $\mathcal{F}$. This set $\{x, y\}$ satisfies $|S \cap \{x, y\}| = 1$. $\square$

3.3.2 Balanced Case: Sparsifying U

If U is k-balanced, then we compute a subset $U' \subseteq U$ of size at most $|U|/2$ using *expander decompositions*, while preserving the condition that U' spans both sides of some Steiner mincut. This section is dedicated to proving the following lemma:

Lemma 3.3.7: Deterministic sparsification of U

Fix any constant $\epsilon > 0$. Then, there is a constant $C = O(1/\epsilon^4)$ such that the following holds. Consider a graph $G = (V, E)$, a parameter $\phi \le 1/(C \log^C n)$, and a set $U \subseteq V$ of vertices that is $(1 + 1/\phi)^3$-balanced with witness (S_1, S_2). Then, we can compute in deterministic $O(m^{1+\epsilon})$ time a set $U' \subseteq U$ with $|U'| \le |U|/2$ such that $S_i \cap U' \ne \emptyset$ for both $i = 1, 2$.

Deterministic expander decomposition. Our main tool will be deterministic expander decompositions with *custom demands*, a generalization of standard expander decompositions. We first introduce some new notation. Let $G = (V, E)$ be a weighted, undirected graph. For disjoint vertex subsets $V_1, \dots, V_\ell \subseteq V$, define $E(V_1, \dots, V_\ell)$ as the set of edges $(u, v) \in E$ with $u \in V_i$ and $v \in V_j$ for some $i \ne j$. Recall that $w(F)$ is the sum of weights of edges in F; i.e., $w(E(V_1, \dots, V_\ell))$ is the sum of weights of edges with endpoints in different vertex sets in $V_1, V_2, \dots, V_\ell$. In particular, for a cut (A, B), we denote the edges in the cut both by $E(A, B)$ as well as the previously introduced notation ∂A (or ∂B), and the weight of the cut is correspondingly denoted $w(E(A, B))$ as well as $w(\partial A)$ (or $w(\partial B)$). For a vector $\mathbf{d} \in \mathbb{R}^V$ of entries on the vertices, define $\mathbf{d}(v)$ as the entry of v in $\mathbf{d}$, and for a subset $U \subseteq V$, define $\mathbf{d}(U) := \sum_{v \in U} \mathbf{d}(v)$.

We now introduce the concept of an expander "weighted" by custom *demands* on the vertices.

Definition 3.3.8: $(\phi, \mathbf{d})$-expander

Consider a weighted, undirected graph $G = (V, E)$ with edge weights w and a vector $\mathbf{d} \in \mathbb{R}^V_{\ge 0}$ of non-negative "demands" on the vertices. The graph G is a $(\phi, \mathbf{d})$-*expander* if for all subsets $S \subseteq V$,

$$\frac{w(\partial S)}{\min\{\mathbf{d}(S), \mathbf{d}(V \setminus S)\}} \ge \phi.$$

Intuitively, to capture the intersection of a set with U, we will place demand λ at each vertex $v \in U$, where λ is the weight of the Steiner mincut, and demand 0 at the remaining vertices. We now state the deterministic algorithm of [27] that computes our desired expander decomposition.

Theorem 3.3.9: Deterministic $(\phi, \mathbf{d})$-expander decomposition

Fix any constant $\epsilon > 0$ and any parameter $\phi > 0$. Given a weighted, undirected graph $G = (V, E)$ with edge weights w and a non-negative demand vector $\mathbf{d} \in \mathbb{R}^V_{\geq 0}$ on the vertices, there is a deterministic algorithm running in $O(m^{1+\epsilon})$ time that partitions V into subsets $V_1, \dots, V_\ell$ such that

1. For each $i \in [\ell]$, define the demands $\mathbf{d}_i \in \mathbb{R}^{V_i}_{\geq 0}$ as $\mathbf{d}_i(v) = \mathbf{d}(v) + w(E(\{v\}, V \setminus V_i))$ for all $v \in V_i$. Then, the graph $G[V_i]$ is a $(\phi, \mathbf{d}_i)$-expander.
2. The total weight $w(E(V_1, \dots, V_\ell))$ of inter-cluster edges is $B\phi\mathbf{d}(V)$ where $B = (\lg n)^{O(1/\epsilon^4)}$.

Sparsification Algorithm. Let $\tilde{\lambda} \in [\lambda, 3\lambda]$ be a 3-approximation to the Steiner mincut value λ, which can be computed in deterministic $\tilde{O}(m)$ time using the $(2+\delta)$-approximation algorithm of Matula (for any $\delta > 0$) [80]. Set $\phi := 1/(C \log^C n)$ for a sufficiently large constant $C > 0$, and let $\epsilon > 0$ be the constant fixed by Theorem 3.3.1. We apply Theorem 3.3.9 to G with parameters ϵ, ϕ and the demand vector $\mathbf{d} \in \mathbb{R}^V_{\geq 0}$ satisfying $\mathbf{d}(v) = \tilde{\lambda}$ for all $v \in U$ and $\mathbf{d}(v) = 0$ for all $v \in V \setminus U$. Observe that $\mathbf{d}(V) = |U| \cdot \tilde{\lambda} \leq |U| \cdot 3\lambda$. Let $V_1, \dots, V_\ell \subseteq V$ be the output, and for each $i \in [\ell]$, define $U_i := V_i \cap U$.

We now describe the procedure to select the subset $U' \subseteq U$. We say that a cluster V_i is *trivial* if $U_i = \emptyset$, *small* if $1 \leq |U_i| \leq 1/\phi^2$, and *large* if $|U_i| > 1/\phi^2$. The algorithm for selecting the set U' is simple:

- for each trivial cluster, do nothing;
- for each small cluster V_i, add an arbitrary vertex of U_i to U';
- for each large cluster V_j, add $1 + 1/\phi$ arbitrary vertices of U_j to U'.

Size bound. First, we prove the desired size bound of the sparsified set U', which is one part of Lemma 3.3.7.

Claim 3.3.10: Number of clusters

There are at most $B\phi|U|$ many clusters; that is, $\ell \leq B\phi|U|$ where $B = (\lg n)^{O(1/\epsilon^4)}$.

Proof. Since λ is the Steiner mincut value of graph G, each cluster V_i has $w(\partial V_i) \geq \lambda$, so the total weight of inter-cluster edges is at least $\ell\lambda/2$. By the guarantee of Theorem 3.3.9, the total weight of inter-cluster edges is at most

$$B\phi\, \mathbf{d}(V) = B\phi|U|\tilde{\lambda} \leq B\phi|U|\lambda,$$

where $B = (\lg n)^{O(1/\epsilon^4)}$. Putting these together gives $\ell \le B\phi|U|$ as desired. □

Corollary 3.3.11: Size bound

There exists a constant $C = O(1/\epsilon^4)$ such that if $\phi \le 1/(C \log^C n)$, then the set U' constructed by the sparsification algorithm satisfies $|U'| \le |U|/2$.

Proof. There are at most $B\phi|U|$ small clusters by Claim 3.3.10. Also, there are at most $\phi^2|U|$ large clusters since each large cluster has at least ϕ^2 vertices in U. This gives

$$\begin{aligned} |U'| &\le B\phi|U| + \phi^2|U| \cdot (1 + 1/\phi) \\ &\le O(B\phi|U|) \le \phi|U| \cdot \frac{C}{2} \log^C n \end{aligned}$$

for an appropriate constant $C = O(B) = O(1/\epsilon^4)$. Since $\phi \le 1/(C \log^C n)$, we have

$$|U'| \le \phi|U| \cdot \frac{C}{2} \log^C n \le |U|/2.$$

□

Hitting both sides of the Steiner mincut. Now, we prove the "hitting" property of the sparsified set U' in Lemma 3.3.7, namely the guarantee that $S_i \cap U' \neq \emptyset$ for both $i = 1, 2$.

The claim below says that the Steiner mincut (A, B) cannot cut too "deeply" into the sets U_i. In particular, if a set U_i is large (say, $|U_i| \gg 1/\phi$), then the Steiner mincut cannot cut U_i evenly in the sense that $|U_i \cap A| \approx |U_i \cap B|$; instead, we either have $|U_i \cap A| \ll |U_i \cap B|$ or $|U_i \cap A| \gg |U_i \cap B|$.

Claim 3.3.12

For any cut (A, B) of G, we have

$$\sum_{i \in [\ell]} \min\{|U_i \cap A|, |U_i \cap B|\} \le \frac{w(E(A, B))}{\phi\lambda},$$

where $U_i := V_i \cap U$ for $i \in [\ell]$.

Proof. Since $G[V_i]$ is a $(\phi, \mathbf{d}_i)$-expander, and since $\mathbf{d}_i(S) \ge \mathbf{d}(S) = |U \cap S| \cdot \tilde{\lambda} \ge |U \cap S| \cdot \lambda$ for all subsets $S \subseteq V_i$, we have

$$\frac{w(E(V_i \cap A, V_i \cap B))}{\min\{|U \cap (V_i \cap A)| \cdot \lambda, |U \cap (V_i \cap B)| \cdot \lambda\}}$$

$$\ge \frac{w(E(V_i \cap A, V_i \cap B))}{\min\{\mathbf{d}_i(U_i \cap A), \mathbf{d}_i(U_i \cap B)\}} \ge \phi.$$

This means that

$$\begin{aligned} &\min\{|U_i \cap A| \cdot \lambda, |U_i \cap B| \cdot \lambda\} \\ &= \min\{|U \cap (V_i \cap A)| \cdot \lambda, |U \cap (V_i \cap B)| \cdot \lambda\} \\ &\leq \frac{w(E(U_i \cap A, U_i \cap B))}{\phi}. \end{aligned}$$

Since $E(V_i \cap A, V_i \cap B)$ is contained in $E(A, B)$ and is disjoint over all i, we have

$$\sum_{i \in [\ell]} w(E(V_i \cap A, V_i \cap B)) \leq w(E(A, B)).$$

Putting things together,

$$\begin{aligned} &\sum_{i \in [\ell]} \min\{|U_i \cap A|, |U_i \cap B|\} \\ &\leq \frac{1}{\lambda} \sum_{i \in [\ell]} \frac{w(E(V_i \cap A, V_i \cap B))}{\phi} \\ &\leq \frac{w(E(A, B))}{\phi\lambda}. \end{aligned}$$

□

We say that a cut C *cuts* a cluster V_i if both $C \cap V_i$ and $V_i \setminus C$ are non-empty. The next claim states that the Steiner mincut can only cut a few clusters V_i, i.e., only a few clusters V_i overlap both sides of the Steiner mincut. This implies that for the sets $U_i \subseteq V_i$ in particular, all but a few of them satisfy $U_i \cap A = \emptyset$ or $U_i \cap B = \emptyset$.

Claim 3.3.13

Let C be one side of a Steiner mincut (i.e., $w(\partial C) = \lambda$). Then, C cuts at most $(1+1/\phi)$ clusters V_i.

Proof. Suppose for contradiction that C cuts more than $(1 + 1/\phi)$ clusters. Fix a cluster V_i that is cut, and let A_i and B_i be $C \cap V_i$ and $V_i \setminus C$ (possibly swapped) so that $w(E(A_i, V \setminus V_i)) \leq w(E(B_i, V \setminus V_i))$. The edges $E(A_i, B_i)$ are contained in ∂C, and across different clusters V_i that are cut, the edges $E(A_i, B_i)$ are disjoint, so

$$\sum_i w(E(A_i, B_i)) \leq w(\partial C) = \lambda.$$

Since C cuts more than $(1 + 1/\phi)$ clusters, there exists a cluster V_i with

$$w(E(A_i, B_i)) < \frac{w(\partial C)}{1 + 1/\phi} = \frac{\lambda}{1 + 1/\phi}.$$

For all subsets $S \subseteq V_i$, we have

$$\mathbf{d}_i(S) \geq \sum_{v \in S} w(E(\{v\}, V \setminus V_i)) = w(E(S, V \setminus V_i)).$$

Since $G[V_i]$ is a $(\phi, \mathbf{d}_i)$-expander,

$$\begin{aligned}
&w(E(A_i, B_i)) \\
&\quad \geq \phi \cdot \min\{\mathbf{d}_i(A_i), \mathbf{d}_i(B_i)\} \\
&\quad \geq \phi \cdot \min\{w(E(A_i, V \setminus V_i)), w(E(B_i.V \setminus V_i))\} \\
&\quad = \phi \cdot w(E(A_i, V \setminus V_i)).
\end{aligned}$$

Consider the cut ∂A_i, which satisfies

$$\begin{aligned}
w(\partial A_i) &= w(E(A_i, B_i)) + w(E(A_i, V \setminus V_i)) \\
&\leq w(E(A_i, B_i)) + \frac{1}{\phi} w(E(A_i, B_i)) \\
&= \left(1 + \frac{1}{\phi}\right) w(E(A_i, B_i)) < \lambda,
\end{aligned}$$

contradicting the fact that C is the Steiner mincut. □

Finally, we prove the "hitting" property of the sparsified set U'. This, along with Corollary 3.3.11, finishes the proof of Lemma 3.3.7.

Lemma 3.3.14: Hitting property

Suppose that U is $(1 + 1/\phi)^3$-balanced with witness (S_1, S_2). Then, for the set U' constructed by the sparsification algorithm, we have $S_i \cap U' \neq \emptyset$ for both $i = 1, 2$.

Proof. For each cluster V_i, by Claim 3.3.12,

$$\min\{|U_i \cap A|, |U_i \cap B|\} \leq \frac{w(E(A, B))}{\phi\lambda} \leq \frac{1}{\phi}.$$

In other words, either $|S_1 \cap U_i| \leq 1/\phi$ or $|S_2 \cap U_i| \leq 1/\phi$. Call a cluster V_i:

1. *white* if $S_1 \cap U_i = \emptyset$ (i.e., $U_i \subseteq S_2$).
2. *light gray* if $0 < |S_1 \cap U_i| \leq |S_2 \cap U_i| < |U_i|$, which implies that $0 < |S_1 \cap U_i| \leq 1/\phi$.
3. *dark gray* if $0 < |S_2 \cap U_i| < |S_1 \cap U_i| < |U_i|$, which implies that $0 < |S_2 \cap U_i| \leq 1/\phi$.
4. *black* if $S_2 \cap U_i = \emptyset$ (i.e., $U_i \subseteq S_1$).

Every cluster must be one of the four colors, and by Claim 3.3.13, there are at most $(1+1/\phi)$ (light or dark) gray clusters since $U_i \cap S_1, U_i \cap S_2 \neq \emptyset$ implies that S_1 cuts cluster V_i. Note that since we are only considering clusters V_i such that $U_i \neq \emptyset$, it must be that for a white cluster, we have $|S_2 \cap U_i| \neq \emptyset$, and similarly, for a black cluster, we have $|S_1 \cap U_i| \neq \emptyset$. There are now a few cases:

1. There are no large clusters. In this case, if there is at least one white and one black small cluster, then the vertices from these clusters added to U' are in S_2 and S_1, respectively. Otherwise, assume w.l.o.g. that there are no black clusters. Since there are at most $(1 + 1/\phi)$ gray clusters in total, $|S_1 \cap U| \leq (1 + 1/\phi) \cdot 1/\phi^2$, contradicting our assumption that $\min\{|S_1 \cap U|, |S_2 \cap U|\} \geq (1 + 1/\phi)^3$.
2. There are large clusters, but all of them are white or light gray. Let V_i be a large white or light gray cluster. Since we select $1 + 1/\phi$ vertices of U_i, and $|S_1 \cap U_i| = \min\{|S_1 \cap U_i|, |S_2 \cap U_i|\} \leq 1/\phi$, we must select at least one vertex not in S_1. Therefore, $S_2 \cap U' \neq \emptyset$. If there is at least one black cluster, then the selected vertex in there is in U', so $S_1 \cap U' \neq \emptyset$ too, and we are done.
   So, assume that there is no black cluster. Since all large clusters are light gray (or white), $|S_1 \cap U_i| \leq 1/\phi$ for all large clusters V_i. Moreover, by definition of small clusters, $|S_1 \cap U_i| \leq |U_i| \leq 1/\phi^2$ for all small clusters V_i. Since there are at most $(1 + 1/\phi)$ gray clusters by Claim 3.3.13,
   $$\begin{aligned} |S_1 \cap U| &= \sum_{i:V_i \text{ small}} |S_1 \cap U_i| + \sum_{i:V_i \text{ large}} |S_1 \cap U_i| \\ &\leq \left(1 + \frac{1}{\phi}\right) \cdot \frac{1}{\phi^2} + \left(1 + \frac{1}{\phi}\right) \cdot \frac{1}{\phi} \\ &= 2\left(1 + \frac{1}{\phi}\right) \cdot \frac{1}{\phi} < \left(1 + \frac{1}{\phi}\right)^3, \end{aligned}$$
   a contradiction.
3. There are large clusters, but all of them are black or dark gray. This is symmetric to case (2) above with S_1 replaced with S_2.
4. There is at least one black or dark gray large cluster V_i, and at least one white or light gray large cluster V_j. In this case, since we select $1 + 1/\phi$ vertices of U_i and $|S_2 \cap U_i| = \min\{|S_1 \cap U_i|, |S_2 \cap U_i|\} \leq 1/\phi$, we must select at least one vertex in S_1. Similarly, we must select at least one vertex in U_j that is in S_2.

□

3.4 Conclusion

In this chapter, we established an equivalence between the Steiner mincut and (s,t)-mincut problems, up to polylogarithmic factors for randomized algorithms and $n^{o(1)}$ factors for deterministic. One immediate question is whether the deterministic reduction can improved to a polylogarithmic overhead. With the current approach, this essentially reduces to whether deterministic expander decompositions can be computed with polylog(n) guarantees everywhere. As previously discussed in Section 8.9, such an improvement would require significantly new ideas. Alternatively, it may be possible to use more powerful tools from de-randomization to bypass expander decompositions altogether.

On the applications side, the Steiner mincut was historically studied in the context of computing a Gomory-Hu tree [16]. It had not seen much action elsewhere due to the prohibitive running time of past algorithms for the problem. Now that Steiner mincut can be solved much more efficiently, we expect more algorithms to use Steiner mincut as an important primitive.

Chapter 4

Gomory-Hu Tree

In this section, we discuss our algorithms for computing a Gomory-Hu tree [71], a classic data structure that encodes all pairwise (s, t)-mincuts in the form of a single tree. That is, an algorithm computing the Gomory-Hu tree can also answer all-pairs (s, t)-mincuts, which is itself a fundamental problem on graphs. We approach the Gomory-Hu tree problem from a locality perspective, combining the isolating cuts lemma (Lemma 2.2.2) and random sampling in a similar way to the Steiner mincut algorithm of Section 3.2.

Our Gomory-Hu tree algorithms resemble Gomory and Hu's original algorithm, which iteratively computes (s, t)-mincuts and applies recursion on each side. However, instead of computing a single (s, t)-mincut at each step, we compute multiple (s, t)-mincuts for a fixed source s, ensuring that we make enough progress at each recursive step. This is where the minimum isolating cuts algorithm becomes useful: it computes an isolating cut around each terminal, giving us many candidate (s, t)-mincuts. However, the difficulty is in verifying which of these cuts are truly (s, t)-mincuts for some t and which are not. In fact, we do not know a verification procedure beyond trivially computing the (s, t)-mincut for each t. Nevertheless, while we do not obtain a faster (exact) Gomory-Hu tree at the end, we are able to reduce the problem down to this verification step, which we call *single-source mincut verification*. This is a significant step towards obtaining faster Gomory-Hu tree algorithms, since future endeavors at the Gomory-Hu tree problem can focus instead on this simpler, seemingly more tractable single-source mincut verification.

If we are happy with an *approximate* Gomory-Hu tree, however, the verification step can be bypassed entirely, leading to an algorithm in roughly max-flow time. To obtain this algorithm, we define a core subroutine called the *Cut Threshold* problem: we are given a source vertex and a real number λ called the cut threshold, and we want to output all vertices whose mincut from the source is at least λ. We solve this problem through minimum isolating cuts in a similar manner as above, and then apply the Cut Threshold subroutine to obtain the approximate Gomory-Hu tree. Since our result, the Cut Threshold problem itself has already seen another application to the edge-augmentation problem [19], suggesting that it is a fundamental problem in its own right.

4.1 Background

Gomory-Hu trees originated from a classic result of Gomory and Hu [44], who showed that by using just $n-1$ max-flows, they could construct a tree T on the vertices of an undirected graph G such that for every pair of vertices s and t, the (s,t) edge connectivity in T was equal to that in G. In other words, the $\binom{n}{2}$ pairs of vertices had at most $n-1$ different edge connectivities and they could be obtained using just $n-1$ max-flow calls.

However, despite rapid advancements in cut and flow algorithms since then, the best algorithm for constructing a GH-tree remains the one given by Gomory and Hu almost six decades after their work. There have been alternatives suggested along the way, although none of them unconditionally improves on the original construction. Bhalgat *et al.* [17] (see also [50]) obtained an $\tilde{O}(mn)$ algorithm for this problem, but only for unweighted graphs, and Abboud *et al.* [2] improved this bound for *sparse* unweighted graphs to $\tilde{O}(m^{3/2}n^{1/6})$.

Due to the intractability of the general Gomory-Hu tree problem, there has been recent action on obtaining *approximate* results. In a beautiful paper, Abboud *et al.* [3] showed that the problem of finding all pairs edge connectivities (that a GH tree obtains) can be reduced to polylog(n) instances of the single source mincut problem: given a fixed source vertex s, find the (s,t)-mincut of s with every other vertex t. They solve the single source mincut problem in $\tilde{O}(n^2)$ time by calling $n-1$ approximate max-flows on an $\tilde{O}(n)$-sized sparsifier of the graph, leading to a total running time of $\tilde{O}(n^2)$. However, they do not recover an approximate Gomory-Hu tree; indeed, prior to our work, no nontrivial result on approximate Gomory-Hu trees were known.

4.2 Our Results

We first study the traditional Gomory-Hu tree problem, defined below.

Definition 4.2.1: Gomory-Hu tree

Given an undirected graph $G=(V,E)$, a Gomory-Hu tree is a weighted tree T on V such that

- For all $s,t \in V$, consider the minimum-weight edge (u,v) on the unique s–t path in T. Let U' be the vertices of the connected component of $T-(u,v)$ containing s. Then, the set $U' \subseteq V$ is an (s,t)-mincut, and its value is the weight of the (u,v) edge in T.

Our first result is a reduction from the Gomory-Hu tree problem to essentially *verifying* a collection of (s,t)-mincuts sharing a common vertex s. We call this problem *single-source mincut verification.*

Definition 4.2.2: Single-source mincut verification

The input to *single-source mincut verification* is a graph $G = (V, E)$, a source vertex $s \in V$, and a value $\tilde{\lambda}_v$ for each $v \in V \setminus s$ such that $\tilde{\lambda}_v \geq \mathsf{mincut}(s, v)$. The task is to determine, for each vertex $v \in V \setminus s$, whether or not $\tilde{\lambda}_v = \mathsf{mincut}(s, v)$.

Theorem 4.2.3: Gomory-Hu tree reduces to single-source mincut verification

There is a randomized algorithm that outputs a Gomory-Hu tree of a weighted, undirected graph w.h.p. It makes calls to single-source mincut verification on graphs with a total of $\tilde{O}(n)$ vertices and $\tilde{O}(m)$ edges, and runs for max-flow time outside of these calls.

Clearly, single-source mincut verification is no harder than computing a Gomory-Hu tree, so the theorem above shows that the two problems are equivalent. Therefore, future efforts at obtaining a faster Gomory-Hu tree algorithm may be directed instead at the single-source mincut verification problem, which seems simpler and more tractable.

Unfortunately, we do not know how to solve single-source mincut verification faster than the $\tilde{O}(mn)$ time algorithm of Bhalgat et al. [16]. However, if we resort to approximations, then we can do better, even on weighted graphs. We define the approximate version of Gomory-Hu tree below.

Definition 4.2.4: Approximate Gomory-Hu tree

Given an undirected graph $G = (V, E)$, a $(1 + \epsilon)$-approximate Gomory-Hu tree is a weighted tree T on V such that

- For all $s, t \in V$, consider the minimum-weight edge (u, v) on the unique s–t path in T. Let U' be the vertices of the connected component of $T - (u, v)$ containing s. Then, the set $U' \subseteq V$ is a $(1 + \epsilon)$-approximate (s, t)-mincut, and its value is the weight of the (u, v) edge in T.

Theorem 4.2.5: Approximate Gomory-Hu tree in max-flow time, weighted

Given a weighted, undirected graph, there is a randomized algorithm that w.h.p., outputs a $(1 + \epsilon)$-approximate Gomory-Hu tree and runs in $\tilde{O}(m)$ time plus calls to exact max-flow on instances with a total of $\tilde{O}(n\epsilon^{-1})$ vertices and $\tilde{O}(n\epsilon^{-1})$ edges. Using the $\tilde{O}(m\sqrt{n})$ time max-flow algorithm of Lee and Sidford [65], the algorithm runs in $\tilde{O}(m + n^{3/2}\epsilon^{-1.5})$ time.

For unweighted graphs, we obtain the following result, which gives a better running time for sparse graphs, assuming state-of-the-art max-flow algorithms.

Theorem 4.2.6: Approximate Gomory-Hu tree in max-flow time, unweighted

Let G be an unweighted, undirected graph. There is a randomized algorithm that w.h.p., outputs a $(1+\epsilon)$-approximate Gomory-Hu tree and runs in $\tilde{O}(m)$ time plus calls to exact max-flow on unweighted instances with a total of $\tilde{O}(n\epsilon^{-1})$ vertices and $\tilde{O}(m\epsilon^{-1})$ edges. Using the $m^{4/3+o(1)}$-time max-flow algorithm for unweighted graphs of Liu and Sidford [75], the algorithm runs in $m^{4/3+o(1)}\epsilon^{-4/3}$ time.

We emphasize that we are reducing an *approximate* problem to calls of *exact* max-flow, which is an important distinction in the proof. It is an open problem whether we can reduce to approximate max-flow instead, which is solvable in near-linear time [96].

The approximate Gomory-Hu tree algorithms solve the following new problem as a subroutine. It looks basic enough that it may have future applications elsewhere, so we include its definition and corresponding result in this introductory section.

Definition 4.2.7: Cut threshold

Given an undirected graph, a source vertex $s \in V$, and a value $\lambda \geq 0$ called the *cut threshold*, the *cut threshold problem* asks to compute all vertices $v \in V \setminus s$ with $\mathsf{mincut}(s, v) \leq \lambda$.

Theorem 4.2.8: Cut Threshold in max-flow time

There is an algorithm solving the cut threshold problem that runs in $\tilde{O}(m)$ time plus polylog(n) calls to max-flow instances on $O(n)$-vertex, $O(m)$-edge graphs.

4.3 Our Techniques

To sketch our main ideas, let us first think of the CT problem (Theorem 4.2.8). Note that this theorem is already sufficient to obtain the improved the running times for the SSMC and APMC problems, although obtaining a $(1+\epsilon)$-approximate GH tree needs additional ideas. To solve the CT problem, our main tool is the *isolating cuts lemma* from Chapter 2. We actually need a slightly stronger version of it in this chapter that guarantees *minimal* $(v, R \setminus v)$-mincuts, along with a disjointness property. The proof is almost unchanged from the original isolating cuts lemma (Lemma 2.2.2), so we skip the proof.

Note that this time, we require calls to (s,t) max-flow instead of (s,t)-mincut. This is because a minimal (s,t)-mincut can be recovered from an (s,t) max-flow in linear time. The change makes no difference in practice, since the state-of-the-art (s,t) max-flow and mincut running times are the same.

We first describe our strategy for the reduction from GH tree to SSMC verification. For now, fix an arbitrary source vertex s. Our goal is to compute a collection of disjoint sets

$S_j \subseteq V$ such that each S_j is an (s,v)-mincut for some v with $|S_j| \le n/2$, and a total size guarantee of $\sum_j |S_j| = \Omega(n/\log n)$. If we can do so, then we can split the graph along the mincuts similarly to the standard recursive GH tree algorithm, obtaining recursive instances of size at most $n - \Omega(n/\log n)$ each. This is enough to obtain a small recursion depth. (Of course, we need to solve a Steiner version of GH tree in the recursive instances, but we leave out that detail in this techniques section.)

Fix a GH tree T of the graph rooted at s, and let v be a vertex for which the subtree T_v of T rooted at v is an (s,v)-mincut with $n_v \le n/2$ vertices. Suppose we sample a set of vertices from $V \setminus s$ at rate $1/n_v$ and define this sample with s added to it as R. Then, we invoke the isolating cuts lemma with this set R, obtaining disjoint sets S_u for each $u \in R$. We call SSMC verification to check whether each set S_u is an (s,u)-mincut, and we separately check whether $|S_u| \le n/2$, and if both checks pass, then we add S_u to our collection of subsets and mark all vertices in S_u as added. How many vertices do we end up marking? For simplicity, let us first ignore the $|S_u| \le n/2$ check. Consider a vertex v and the subtree T_v of T rooted at v. With constant probability, exactly one vertex from T_v is sampled in R, and with probability $\Omega(1/n_v)$, this sampled vertex is v itself. In that happens, the isolating cuts lemma would return the $s-v$ mincut, namely the cut represented by the edge (u,v) in the GH tree, and this cut S_v would pass the SSMC verification check. This allows us to mark the n_v vertices in T_v. So, roughly speaking, we are able to mark n_v vertices with probability $1/n_v$ in this case, so on average, a vertex in T_v is marked with at least constant probability. Of course, we do not know the value of n_v, but we separately try all sampling levels in inverse powers of 2 and choose the sampling level with the most marked vertices. We formalize and refine this argument to show that there is a sampling level for which we can indeed mark $\Omega(n/\log n)$ vertices in expectation. Finally, we use an argument from [3] that if $s \in V$ is chosen *uniformly at random*, then the additional $|S_u| \le n/2$ check still allows us to mark $\Omega(n/\log n)$ vertices in expectation.

The algorithm for the cut threshold (CT) problem is similar, except that instead of performing the $|S_u| \le n/2$ and SSMC verification checks, we simply check whether $w(\partial S_u) \le \lambda$. We then use the CT algorithm as a "sieve" to obtain an approximate SSMC algorithm for a given source s, which is to compute a $(1+\epsilon)$-approximate (s,v)-mincut for each $v \in V \setminus s$. We start with $\mathsf{mincut}(s,v)$ for all vertices $v \in V \setminus \{s\}$ tentatively set to the maximum possible edge connectivity (call it $\lambda_{\max}$). Next, we run the CT algorithm with $\lambda = (1-\epsilon)\lambda_{\max}$. The vertices v that are identified by this algorithm as having $\mathsf{mincut}(s,v) \le \lambda$ drop down to the next level of the hierarchy, while the remaining vertices v' are declared to have $\mathsf{mincut}(s,v') \in ((1-\epsilon)\lambda, \lambda]$. In the next level of the hierarchy, we again invoke the CT algorithm, but now with λ equal to $(1-\epsilon)$ factor of the previous iteration. In this manner, we iteratively continue moving down the hierarchy, cutting the threshold λ by a factor of $(1-\epsilon)$ in every step, until the connectivity of all vertices has been determined.

Finally, we come to the problem of obtaining an approximate GH tree. The major difficulty we face, compared to the exact GH tree case, is controlling the errors that propagate in the recursive algorithm. Gomory and Hu's original algorithm uses the following strategy: find an (s,t)-mincut for any pair of vertices s and t, and recurse on the two sides of the cut in separate subproblems where the other side of the cut is contracted to a single vertex. They used submodularity of cuts to show that contracting one side of an (s,t)-mincut does not change the connectivity between vertices on the other side. Moreover, they gave a procedure for combining the two GH trees returned by the recursive calls into a single GH tree at the end of the recursion. Ideally, we would like to use the same algorithm but replace an exact (s,t)-mincut with an approximate one. But now, the connectivities in the recursive subproblems are (additively) distorted by the approximation error of the (s,t)-mincut. This imposes two additional restrictions. **(a)** First, the values of the (s,t)-mincuts identified in the recursive algorithm must now be monotone non-decreasing with depth of the recursion so that the approximation error on a larger (s,t)-mincut doesn't get propagated to a smaller $s'-t'$ mincut further down in the recursion. **(b)** Second, the depth of recursion must now be $\text{polylog}(n)$ so that one can control the buildup of approximation error in the recursion by setting the error parameter in a single step to be $\epsilon/\text{polylog}(n)$. Unfortunately, neither of these conditions is met by Gomory and Hu's algorithm. For instance, the recursion depth can be $n-1$ if each (s,t)-mincut is a degree cut. The order of (s,t)-mincut values in the recursion is also arbitrary and depends on the choice of s and t in each step (which itself is arbitrary).

Let us first consider condition **(a)**. Instead of finding the (s,t)-mincut for an arbitrary pair of terminal vertices s and t, suppose we found the Steiner mincut on the terminals, i.e., the cut of smallest value that splits the terminals. This would also suffice in terms of the framework since a Steiner mincut is also an (s,t)-mincut for some pair s,t. But, it brings additional advantages: namely, we get the monotonicity in cut values with recursive depth that we desire. At a high level, this is the idea that we implement: we use the CT algorithm (with some technical modifications) where we set the threshold λ to the value of the Steiner mincut, and identify a partitioning of the terminals where each subset of the partition represents a $(1+\epsilon)$ approximation to the Steiner mincut.

But, how do we achieve condition **(b)**? Fixing the vertex s in the invocation of the SSMC algorithm, we can identify terminal vertices v that have $\mathsf{mincut}(s,v) \in ((1-\epsilon)\lambda, \lambda]$, where λ is the Steiner mincut. But, these approximate Steiner mincuts might be unbalanced in terms of the number of vertices on the two sides of the cut. To understand the problem, suppose there is a single Steiner mincut identified by the CT algorithm, and this cut is the degree cut of s. Then, one subproblem contains all but one vertex in the next round of recursion; consequently, the recursive depth can be high. We overcome this difficulty in two steps. First, we ensure that the only "large" subproblem that we recurse on is the one that contains s. This can be ensured by sampling $O(\log n)$ different vertices as s, which boosts the probability that s is on the larger side of an unbalanced approximate Steiner mincut. This ensures that in the recursion tree, we can only have a large recursive depth along the path containing s. Next, we show that even though we are using an approximate method for detemining mincuts, the approximation error only distorts the connectivities in the subproblems not containing s. This ensures that the approximation errors can build up only along paths in the recursion tree that have depth $O(\log n)$. Combining these two techniques, we obtain our overall algorithm for an approximate GH tree.

4.4 Additional Preliminaries

For our algorithm, it is more convenient to work with Gomory-Hu *Steiner* trees, which are more amenable to our recursive structure.

Definition 4.4.1: Gomory-Hu Steiner tree

Given a graph $G = (V, E)$ and a set of terminals $U \subseteq V$, the Gomory-Hu Steiner tree is a weighted tree T on the vertices U, together with a function $f : V \to U$, such that

- For all $s, t \in U$, consider the minimum-weight edge (u, v) on the unique s–t path in T. Let U' be the vertices of the connected component of $T - (u, v)$ containing s. Then, the set $f^{-1}(U') \subseteq V$ is an (s, t)-mincut, and its value is $w_T(u, v)$.

Definition 4.4.2: Approximate Gomory-Hu Steiner tree

Given a graph $G = (V, E)$ and a set of terminals $U \subseteq V$, the $(1 + \epsilon)$-approximate Gomory-Hu Steiner tree is a weighted tree T on the vertices U, together with a function $f : V \to U$, such that

- For all $s, t \in U$, consider the minimum-weight edge (u, v) on the unique s–t path in T. Let U' be the vertices of the connected component of $T - (u, v)$ containing s. Then, the set $f^{-1}(U') \subseteq V$ is a $(1 + \epsilon)$-approximate (s, t)-mincut, and its value is $w_T(u, v)$.

In our analysis, we use the notion of a *minimal* Gomory-Hu tree. We define this next.

Definition 4.4.3: Rooted minimal Gomory-Hu Steiner tree

Given a graph $G = (V, E)$ and a set of terminals $U \subseteq V$, a rooted minimal Gomory-Hu Steiner tree is a Gomory-Hu Steiner tree on U, rooted at some vertex $r \in U$, with the following additional property:

($*$) For all $t \in U \setminus \{r\}$, consider the minimum-weight edge (u, v) on the unique $r - t$ path in T; if there are multiple minimum weight edges, let (u, v) denote the one that is *closest to t*. Let U' be the vertices of the connected component of $T - (u, v)$ containing r. Then, $\partial_G f^{-1}(U') \subseteq V$ is a *minimal* (r, t)-mincut, and its value is $w_T(u, v)$.

The following theorem establishes the existence of a rooted minimal Gomory-Hu Steiner tree rooted at any given vertex.

Theorem 4.4.4: Existence of rooted minimal Gomory-Hu Steiner tree

For any graph $G = (V, E)$, terminals $U \subseteq V$, and root $r \in U$, there exists a rooted minimal Gomory-Hu Steiner tree rooted at r.

Proof. Let $\epsilon > 0$ be a small enough weight, and let G' be the graph G with an additional edge (r, v) of weight ϵ added for each $v \in V \setminus \{r\}$. (If the edge (r, v) already exists in G, then increase its weight by ϵ instead.) If $\epsilon > 0$ is small enough, then for all $t \in V \setminus \{r\}$ and $S \subseteq V$, if $\partial_{G'} S$ is an (r, t)-mincut in G', then $\partial_G S$ is an (r, t)-mincut in G.

Let (T', f) be a Gomory-Hu Steiner tree for G'. We claim that it is essentially a minimal Gomory-Hu Steiner tree for G, except that its edge weights need to be recomputed as mincuts in G and not G'. More formally, let T be the tree T' with the following edge re-weighting: for each edge (u, v) in T, take a connected component U' of $T - (u, v)$ and reset the edge weight of (u, v) to be $w(\partial_G f^{-1}(U'))$ and not $w(\partial_{G'} f^{-1}(U'))$. We now claim that (T, f) is a minimal Steiner Gomory-Hu tree for G.

We first show that (T, f) is a Gomory-Hu Steiner tree for G. Fix $s, t \in U$, let (u, v) be the minimum-weight edge on the s–t path in T', and let U' be the vertices of the connected component of $T' - (u, v)$ containing s. Since (T', f) is a Gomory-Hu Steiner tree for G', we have that $\partial_{G'} f^{-1}(U')$ is an (s, t)-mincut in G'. If $\epsilon > 0$ is small enough, then by our argument from before, $\partial_G f^{-1}(U')$ is also an (s, t)-mincut in G. By our edge re-weighting of T, the edge (u, v) has the correct weight. Moreover, (u, v) is the minimum-weight edge on the s–t path in T, since a smaller weight edge would contradict the fact that $\partial_G f^{-1}(U')$ is an (s, t)-mincut.

We now show the additional property $(*)$ that makes (T, f) a minimal Gomory-Hu Steiner tree. Fix $t \in U \setminus \{r\}$, and let (u, v) and U' be defined as in $(*)$, i.e., (u, v) is the minimum-weight edge (u, v) on the $r - t$ path that is closest to t, and U' is the vertices of the connected component of $T - (u, v)$ containing r. Since (T, f) is a Gomory-Hu Steiner tree for G, we have that $\partial_G f^{-1}(U')$ is an (r, t)-mincut of value $w_T(u, v)$. Suppose for contradiction that $\partial_G f^{-1}(U')$ is not a *minimal* (r, t)-mincut. Then, there exists $S \subsetneq f^{-1}(U')$ such that ∂S is also an (r, t)-mincut. By construction of G', $w(\partial_{G'} S) = w(\partial_G S) + |S|\epsilon$ and $w(\partial_{G'} f^{-1}(U')) = w(\partial_G f^{-1}(U')) + |f^{-1}(U')|\epsilon$. We have $w(\partial_G S) = w(\partial_G f^{-1}(U'))$ and $|S| < |f^{-1}(U')|$, so $w(\partial_{G'} S) < w(\partial_{G'} f^{-1}(U'))$. In other words, $f^{-1}(U')$ is not an (r, t)-mincut in G', contradicting the fact that (T', f) is a Gomory-Hu Steiner tree for G'. Therefore, property $(*)$ is satisfied, concluding the proof. □

4.5 Reducing to SSMC Verification

In this section, we prove Theorem 4.2.3. The GH tree algorithm is described in Algorithm GHTree a few pages down.

4.5.1 A Single Recursive Step

Before we present Algorithm GHTree, we first consider the subprocedure GHTreeStep that it uses on each recursive step.

Algorithm 2 GHTreeStep($G = (V, E), s, U$)

```
1. Initialize $R^0 \leftarrow U$ and $D \leftarrow \emptyset$
2. For all $i$ from 0 to $\lfloor \lg |U| \rfloor$ do:
    (a) Call Lemma 2.2.2 on $T = R^i$, obtaining disjoint sets $S^i_v$ (the minimal $(v, R^i \setminus v)$-mincut) for each $v \in R^i$.
    (b) Call single-source mincut verification on graph $G$, source $s$, and values $\tilde{\lambda}_v = w(\partial S^i_v)$ for $v \in R^i$. (We do not care about any $v \notin R^i$, so we can set $\tilde{\lambda}_v = \infty$ for them.)
    (c) Let $D^i \subseteq R^i$ be the union of $S^i_v \cap U$ over all $v \in R^i \setminus \{s\}$ satisfying $\tilde{\lambda}_v = \mathsf{mincut}(s, v)$ and $|S^i_v \cap U| \leq |U|/2$
    (d) $R^{i+1} \leftarrow$ subsample of $R^i$ where each vertex in $R^i \setminus \{s\}$ is sampled independently with probability $1/2$, and $s$ is sampled with probability 1
3. Return the largest set $D^i$ and the corresponding sets $S^i_v$ over all $v \in R^i \setminus \{s\}$ satisfying the conditions in line 2c
```

Let $D = D^0 \cup D^1 \cup \cdots \cup D^{\lfloor \lg |U| \rfloor}$ be the union of the sets D^i as defined in Algorithm GHTreeStep. Let D^* be all vertices $v \in U \setminus \{s\}$ for which there exists an (s, v)-mincut whose v side has at most $|U|/2$ vertices in U. We now claim that D covers a large fraction of vertices in D^* in expectation.

Lemma 4.5.1: D covers a large fraction of D^* in expectation

$\mathbb{E}[|D \cap D^*|] = \Omega(|D^*|/\log |U|)$.

Proof. Consider a rooted minimal Steiner Gomory-Hu tree T of G on terminals U rooted at s, which exists by Theorem 4.4.4. For each vertex $v \in U \setminus \{s\}$, let $r(v)$ be defined as the child vertex of the lowest weight edge on the path from v to s in T. If there are multiple lowest weight edges, choose the one with the maximum depth.

For each vertex $v \in D^*$, consider the subtree rooted at v, define U_v to be the vertices in the subtree, and define n_v as the number of vertices in the subtree. We say that a vertex $v \in D^*$ is *active* if $v \in R^i$ for $i = \lfloor \lg n_{r(v)} \rfloor$. In addition, if $U_{r(v)} \cap R^i = \{v\}$, then we say that v *hits* all of the vertices in $U_{r(v)}$ (including itself); see Figure 4.1. In particular, in order for v to hit any other vertex, it must be active. For completeness, we say that any vertex in $U \setminus D^*$ is not active and does not hit any vertex.

To prove that $\mathbb{E}[|D|] \geq \Omega(|D^*|/\log |U|)$, we will show that

(a) each vertex u that is hit is in D,

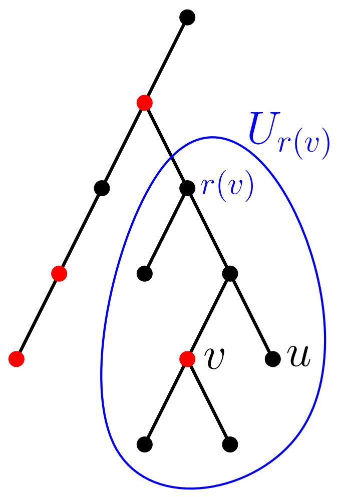

Figure 4.1: Let $i = \lfloor \lg n_{r(v)} \rfloor = \lfloor \lg 7 \rfloor = 2$, and let the red vertices be those sampled in R^2. Vertex v is active and hits u because v is the only vertex in $U_{r(v)}$ that is red.

(b) the total number of pairs (u, v) for which $v \in D^*$ hits u is at least $c|D^*|$ in expectation for some small enough constant $c > 0$, and

(c) with probability at least $1 - \frac{c}{2|U|^2}$ (for the constant $c > 0$ in (b)), each vertex u is hit by at most $O(\log |U|)$ vertices $v \in D^*$.

For (a), consider the vertex v that hits u. By definition, for $i = \lfloor \lg n_{r(v)} \rfloor$, we have $U_{r(v)} \cap R^i = \{v\}$, so $\partial f^{-1}(U_{r(v)})$ is a $(v, R^i \setminus \{v\})$-cut. By the definition of $r(v)$, we have that $\partial f^{-1}(U_{r(v)})$ is a (v, s)-mincut. On the other hand, we have that ∂S_v^i is a $(v, R^i \setminus \{v\})$-mincut, so in particular, it is a (v, s)-cut. It follows that $\partial f^{-1}(U_{r(v)})$ and ∂S_v^i are both (v, s)-mincuts and $(v, R^i \setminus v)$-mincuts, and $w(\partial S_v^i) = \mathsf{mincut}(s, v) \leq W$. Since T is a minimal Gomory-Hu Steiner tree, we must have $f^{-1}(U_{r(v)}) \subseteq S_v^i$. Since S_v^i is the minimal $(v, R^i \setminus \{v\})$-mincut, it is also the minimal (v, s)-mincut, so $S_v^i \subseteq f^{-1}(U_{r(v)})$. It follows that $f^{-1}(U_{r(v)}) = S_v^i$. Since $f^{-1}(U_{r(v)})$ is the minimal (v, s)-mincut and $v \in D^*$, we must have $|f^{-1}(U_{r(v)}) \cap U| \leq z$, so in particular, $|S_v^i \cap U| = |f^{-1}(U_{r(v)}) \cap U| \leq z$. Therefore, the vertex v satisfies all the conditions of line 2c. Moreover, since $u \in U_{r(v)} \subseteq f^{-1}(U_{r(v)}) = S_v^i$, vertex u is added to D in the set $S_v^i \cap U$.

For (b), for $i = \lfloor \lg n_{r(v)} \rfloor$, we have $v \in R^i$ with probability exactly $1/2^i = \Theta(1/n_{r(v)})$, and with probability $\Omega(1)$, no other vertex in $U_{r(v)}$ joins R^i. Therefore, v is active with probability $\Omega(1/n_{r(v)})$. Conditioned on v being active, it hits exactly $n_{r(v)}$ many vertices. It follows that v hits $\Omega(1)$ vertices in expectation.

For (c), the number of vertices v that hit vertex u is at most the number of active vertices v for which $r(v)$ is on the path from u to s in T. Label these vertices $u = v_1, v_2, \ldots, v_\ell = s$, ordered by increasing distance from u to $r(v_i)$ in T. Each vertex $v_j \in D^*$ is active with probability $\Theta(1/n_{r(v_j)})$, which is at most $\Theta(1/j)$ since $v_1, \ldots, v_j \in U_{r(v_j)}$. Each vertex $v_j \notin D^*$ is never active. Therefore, the expected number of active vertices on the path from u to s

is at most $\sum_{j=1}^{\ell} \Theta(1/j) = \Theta(\ln \ell) \leq \Theta(\ln |U|)$. A standard Chernoff bound shows that with probability at least $1 - \frac{c}{2|U|^3}$ for any constant $c > 0$, the number of active vertices on the path is indeed $O(\ln |U|)$, where the $O(\cdot)$ hides the dependency on c. Taking a union bound over all $u \in U$, the probability that this is true for all vertices is at least $1 - \frac{c}{2|U|^2}$.

Finally, we show why properties (a) to (c) imply $\mathbb{E}[|D|] \geq \Omega(|D^*|/\log |U|)$. In the event that property (c) fails, the total number of pairs (u, v) for which v hits u can be trivially upper bounded by $|U|^2$. Since this occurs with probability at most $\frac{c}{2|U|^2}$, the total contribution to the expectation $c|D^*|$ in property (b) is at most $c/2$. Therefore, the contribution to the expectation in the event that property (c) succeeds is at least $c|D^*| - c/2 \geq (c/2)|D^*|$. In this case, since each vertex is hit at most $O(\log |U|)$ times, there are at least $\Omega(|D^*|/\log |U|)$ vertices hit in expectation. □

The corollary below immediately follows. Note that the sets S_v^i output by the algorithm are disjoint, which we require in the recursive GH tree algorithm.

Corollary 4.5.2: The largest D^i covers a large fraction of D^* in expectation

The largest set D^i returned by GHTreeStep satisfies $\mathbb{E}[|D^i \cap D^*|] = \Omega(|D^*|/\log^2 |U|)$.

4.5.2 The Gomory-Hu Tree Algorithm

The Gomory-Hu tree algorithm is presented in GHTree, which uses GHTreeStep as a subprocedure on each recursive step.

Correctness. Algorithm GHTree has the same recursive structure as Gomory and Hu's original algorithm, except that it computes multiple mincuts on each step. Therefore, correctness of the algorithm follows similarly to their analysis. For completeness, we include it below.

Lemma 4.5.3: Correctness of Algorithm GHTree

Algorithm GHTree$(G = (V, E), U)$ outputs a Gomory-Hu Steiner tree.

To prove Lemma 4.5.3, we first introduce a helper lemma.

Lemma 4.5.4: Mincuts in G_{large} and mincuts in U_{large} are preserved exactly

For any distinct vertices $p, q \in U_{\text{large}}$, we have $\mathsf{mincut}_{G_{\text{large}}}(p, q) = \mathsf{mincut}_G(p, q)$. The same holds with U_{large} and G_{large} replaced by U_v and G_v for any $v \in D^i$.

Proof. Since G_{large} is a contraction of G, we have $\mathsf{mincut}_{G_{\text{large}}}(p, q) \geq \mathsf{mincut}_G(p, q)$. To show the reverse inequality, fix any (p, q)-mincut in G, and let S be one side of the mincut. We show that for each $v \in R^i$, either $S_v^i \subseteq S$ or $S_v^i \subseteq V \setminus S$. Assuming this, the cut ∂S stays intact when the sets S_v^i are contracted to form G_{large}, so $\mathsf{mincut}_{G_{\text{large}}}(p, q) \leq w(\partial S) = \mathsf{mincut}_G(p, q)$.

Algorithm 3 GHTREE$(G = (V, E), U)$

```
1. $s \leftarrow$ uniformly random vertex in $U$
2. Call GHTREESTEP$(G, s, U)$ to obtain $D^i$ and the sets $S^i_v$ (so that $D^i = \bigcup S^i_v \cap U$)

3. For each set $S^i_v$ do:        ▷ Construct recursive graphs and apply recursion
   (a) Let $G_v$ be the graph $G$ with vertices $V \setminus S^i_v$ contracted to a single vertex $x_v$ ▷ $S^i_v$ are disjoint
   (b) Let $U_v \leftarrow S^i_v \cap U$
   (c) If $|U_v| > 1$, then recursively set $(T_v, f_v) \leftarrow$ GHTREE$(G_v, U_v)$
4. Let $G_{\text{large}}$ be the graph $G$ with (disjoint) vertex sets $S^i_v$ contracted to single vertices $y_v$ for all $v \in D^i$
5. Let $U_{\text{large}} \leftarrow U \setminus D^i$
6. If $|U_v| > 1$, then recursively set $(T_{\text{large}}, f_{\text{large}}) \leftarrow$ GHTREE$(G_{\text{large}}, U_{\text{large}})$
7. Combine $(T_{\text{large}}, f_{\text{large}})$ and $\{(T_v, f_v) : v \in D^i\}$ into $(T, f)$ according to COMBINE
8. Return $(T, f)$
```

Algorithm 4 COMBINE$((T_{\text{large}}, f_{\text{large}}), \{(T_v, f_v) : v \in R^i\})$

```
1: Construct $T$ by starting with the disjoint union $T_{\text{large}} \cup \bigcup_{v \in R^i} T_v$ and, for each $v \in R^i$, adding an edge between $f_v(x_v) \in U_v$ and $f_{\text{large}}(y_v) \in U_{\text{large}}$ of weight $w(\partial_G S^i_v)$
2: Construct $f : V \to U$ by $f(v') = f_{\text{large}}(v')$ if $v' \in U_{\text{large}}$ and $f(v') = f_v(v')$ if $v' \in U_v$ for some $v \in R^i$
3: **return** $(T, f)$
```

Consider any $v \in R^i$, and suppose first that $v \in S$. Then, $S_v^i \cap S$ is still a $(v, R^i \setminus v)$-cut, and $S_v^i \cup S$ is still a (p, q)-cut. By the submodularity of cuts,

$$w(\partial_G S_v^i) + w(\partial_G S) \geq w(\partial_G (S_v^i \cup S)) + w(\partial_G (S_v^i \cap S)).$$

In particular, $S_v^i \cap S$ must be a minimum $(v, R^i \setminus v)$-cut. Since S_v^i is the minimal $(v, R^i \setminus v)$-mincut, it follows that $S_v^i \cap S = S_v^i$, or equivalently, $S_v^i \subseteq S$.

Suppose now that $v \notin S$. In this case, we can swap p and q, and swap S and $V \setminus S$, and repeat the above argument to get $S_v^i \subseteq V \setminus S$.

The argument for U_v and G_v is identical, and we skip the details. □

Proof (Lemma 4.5.3). We apply induction on $|U|$. By induction, the recursive outputs $(T_{\text{large}}, f_{\text{large}})$ and (T_v, f_v) are Gomory-Hu Steiner trees. By definition, this means that for all $x, y \in U_{\text{large}}$ and the minimum-weight edge (u, u') on the x–y path in T_{large}, letting $U'_{\text{large}} \subseteq U_{\text{large}}$ be the vertices of the connected component of $T_{\text{large}} - (u, u')$ containing x, we have that $f_{\text{large}}^{-1}(U'_{\text{large}})$ is an (s, t)-mincut in G_{large} with value is $w_T(u, u')$. Define $U' \subseteq U$ as the vertices of the connected component of $T - (u, u')$ containing x. By construction of (T, f) (lines 1 and 2), the set $f^{-1}(U')$ is simply $f_{\text{large}}^{-1}(U'_{\text{large}})$ with the vertex x_{large} replaced by $V \setminus S_{\text{large}}^i$ in the case that $x_{\text{large}} \in f^{-1}(U')$. Since G_{large} is simply G with all vertices $V \setminus S_{\text{large}}^i$ contracted to x_{large}, we conclude that $w_{G_{\text{large}}}(\partial f_{\text{large}}^{-1}(U'_{\text{large}})) = w_G(\partial f^{-1}(U'))$. By Lemma 4.5.4, we have $\mathsf{mincut}_G(x, y) = \mathsf{mincut}_{G_{\text{large}}}(x, y)$ are equal, so $w_G(\partial f^{-1}(U'))$ is an (x, y)-mincut in G. In other words, the Gomory-Hu Steiner tree condition for (T, f) is satisfied for all $x, y \in U_{\text{large}}$. A similar argument handles the case $x, y \in U_v$ for some $v \in R^i$.

There are two remaining cases: $x \in U_v$ and $y \in U_{\text{large}}$, and $x \in U_v$ and $y \in U_{v'}$ for distinct $v, v' \in R^i$. Suppose first that $x \in U_v$ and $y \in U_{\text{large}}$. By considering which sides v and s lie on the (x, y)-mincut, we have

$$w(\partial_G S) = \mathsf{mincut}(x, y) \geq \min\{\mathsf{mincut}(x, v), \mathsf{mincut}(v, s), \mathsf{mincut}(s, y)\}.$$

We now case on which of the three mincut values $\mathsf{mincut}(x, y)$ is greater than or equal to.

1. If $\mathsf{mincut}(x, y) \geq \mathsf{mincut}(v, s)$, then since S_v^i is a (v, s)-mincut that is also an (x, y)-cut, we have $\mathsf{mincut}(x, y) = \mathsf{mincut}(v, s)$. By construction, the edge $(f_v(x_v), f_{\text{large}}(y_v))$ of weight $w(\partial_G S_v^i) = w(\partial_G S)$ is on the $x - y$ path in T. There cannot be edges on the $x - t$ path in T of smaller weight, since each edge corresponds to a (s, t)-cut in G of the same weight. Therefore, $(f_v(x_v), f_{\text{large}}(y_v))$ is the minimum-weight edge on the s–t path in T.
2. Suppose now that $\mathsf{mincut}(x, v) \leq \mathsf{mincut}(x, y) < \mathsf{mincut}(v, s)$. The minimum-weight edge e on the $x - v$ path in T_v has weight $\mathsf{mincut}(x, v)$. This edge e cannot be on the v–$f_v(x_v)$ path in T_v, since otherwise, we would obtain a (v, x_v)-cut of value $\mathsf{mincut}(x, v)$ in G_v, which becomes a (v, s)-cut in G after expanding the contracted vertex x_v; this contradicts our assumption that $\mathsf{mincut}(x, v) < \mathsf{mincut}(v, s)$. It follows that e is on the $x - f_v(x_v)$ path in T_v which, by construction, is also on the $x - y$ path in T. Once

again, the $x - y$ path cannot contain an edge of smaller weight.

3. The final case $\mathsf{mincut}(s, y) \le \mathsf{mincut}(x, y) < \mathsf{mincut}(v, s)$ is symmetric to case 2, except we argue on T_{large} and G_{large} instead of T_v and G_v.

Suppose now that $x \in U_v$ and $y \in U_{v'}$ for distinct $v, v' \in R^i$. By considering which sides v, v', s lie on the (x, y)-mincut, we have

$$w(\partial_G S) = \mathsf{mincut}(x, y) \ge \min\{\mathsf{mincut}(x, v), \mathsf{mincut}(v, s), \mathsf{mincut}(s, v'), \mathsf{mincut}(v', y)\}.$$

We now case on which of the four mincut values $\mathsf{mincut}(x, y)$ is greater than or equal to.

1. If $\mathsf{mincut}(x, y) \ge \mathsf{mincut}(v, s)$ or $\mathsf{mincut}(x, y) \ge \mathsf{mincut}(s, v')$, then the argument is the same as case 1 above.
2. If $\mathsf{mincut}(x, v) \le \mathsf{mincut}(x, y) < \mathsf{mincut}(v, s)$ or $\mathsf{mincut}(y, v') \le \mathsf{mincut}(x, y) < \mathsf{mincut}(v', s)$, then the argument is the same as case 2 above.

This concludes all cases, and hence the proof. □

Lemma 4.5.5: Recursion depth

W.h.p., the algorithm GHTree has maximum recursion depth $O(\log^3 n)$.

Proof. By construction, each recursive instance (G_v, U_v) has $|U_v| \le |U|/2$. We use the following lemma from [3].

Lemma 4.5.6: Random selection of s [3]

Suppose the source vertex $s \in U$ is chosen uniformly at random. Then, $\mathbb{E}[|D^*|] = \Omega(|U| - 1)$.

By Corollary 4.5.2 and Lemma 4.5.6, over the randomness of s and GHTreeStep, we have

$$\mathbb{E}[D^i] \ge \Omega(\mathbb{E}[|D^*|]/\log^2 |U|) \ge \Omega((|U| - 1)/\log^2 |U|),$$

so the recursive instance $(G_{\text{large}}, U_{\text{large}})$ satisfies $\mathbb{E}[|U_{\text{large}}|] \le (1 - 1/\log^2 |U|) \cdot (|U| - 1)$. Therefore, each recursive branch either has at most half the vertices in U, or has at most a $(1 - 1/\log^2 |U|)$ fraction in expectation. It follows that w.h.p., all branches terminate by $O(\log^3 n)$ recursive calls. □

Lemma 4.5.7: Running time

For an unweighted/weighted graph $G = (V, E)$, and terminals $U \subseteq V$, GHTree(G, V) takes time $\tilde{O}(m)$ plus calls to max-flow on unweighted/weighted instances with a total of $\tilde{O}(n)$ vertices and $\tilde{O}(m)$ edges.

Proof. For a given recursion level, consider the instances $\{(G_i, U_i, W_i)\}$ across that level. By construction, the terminals U_i partition U. Moreover, the total number of vertices over all

G_i is at most $n + 2(|U| - 1) = O(n)$ since each branch creates 2 new vertices and there are at most $|U| - 1$ branches.

To bound the total number of edges, we consider the unweighted and weighted cases separately, starting with the unweighted case. The total number of new edges created is at most the sum of weights of the edges in the final $(1 + \epsilon)$-approximate Gomory-Hu Steiner tree. For an unweighted graph, this is $O(m)$ by the following well-known argument. Root the Gomory-Hu Steiner tree T at any vertex $r \in U$; for any $v \in U \setminus r$ with parent u, the cut $\partial\{v\}$ in G is a (u, v)-cut of value $\deg(v)$, so $w_T(u, v) \leq \deg(v)$. Overall, the sum of the edge weights in T is at most $\sum_{v \in U} \deg(v) \leq 2m$.

For the weighted case, define a *parent* vertex in an instance as a vertex resulting from either (1) contracting $V \setminus S_v^i$ in some previous recursive G_v call, or (2) contracting a component containing a parent vertex in some previous recursive call. There are at most $O(\log n)$ parent vertices: at most $O(\log n)$ can be created by (1) since each G_v call decreases $|U|$ by a constant factor, and (2) cannot increase the number of parent vertices. Therefore, the total number of edges adjacent to parent vertices is at most $O(\log n)$ times the number of vertices. Since there are $O(n)$ vertices in a given recursion level, the total number of edges adjacent to parent vertices is $O(n \log n)$ in this level. Next, we bound the number of edges not adjacent to a parent vertex by m. To do so, we first show that on each instance, the total number of these edges over all recursive calls produced by this instance is at most the total number of such edges in this instance. Let $P \subseteq V$ be the parent vertices; then, each G_v call has exactly $|E(G[S_v^i \setminus P])|$ edges not adjacent to parent vertices (in the recursive instance), and the G_{large} call has at most $|E(G[V \setminus P]) \setminus \bigcup_v E(G[S_v^i \setminus P])|$, and these sum to $|E(G[V \setminus P])|$, as promised. This implies that the total number of edges not adjacent to a parent vertex at the next level is at most the total number at the previous level. Since the total number at the first level is m, the bound follows.

Therefore, there are $O(n)$ vertices and $\tilde{O}(m)$ edges in each recursion level. By Lemma 4.5.5, there are $O(\epsilon^{-1} \log^4 n)$ levels (since $\Delta = 1$ for an unweighted graph), for a total of $\tilde{O}(n\epsilon^{-1})$ vertices and $\tilde{O}(m\epsilon^{-1})$ edges. In particular, the instances to the max-flow calls have $\tilde{O}(n\epsilon^{-1})$ vertices and $\tilde{O}(m\epsilon^{-1})$ edges in total. □

4.6 The Cut Threshold Algorithm

We now present the cut threshold (CT) algorithm, an important building block for the approximate GH tree algorithm. We first describe a single step of the CutThreshold algorithm (we call this CutThresholdStep).

We remark that throughout this section, we will always set $z = \infty$, so the constraint $|S_v^i \cap U| \leq z$ in line 4 can be ignored. However, the variable z will play a role in the next section on computing a Gomory-Hu tree.

Let $D = D^0 \cup D^1 \cup \cdots \cup D^{\lfloor \lg |U| \rfloor}$ be the union of the sets output by the algorithm. Let D^* be all vertices $v \in U \setminus s$ for which there exists an (s, v)-cut of weight at most W whose

Algorithm 5 CutThresholdStep($G = (V, E), s, U, W, z$)

```
1: Initialize $R^0 \leftarrow U$ and $D \leftarrow \emptyset$
2: for $i$ from 0 to $\lfloor \lg |U| \rfloor$ do
3:     Compute minimum isolating cuts $\{S_v^i : v \in R^i\}$ on inputs $G$ and $R^i$
4:     Let $D^i$ be the union of $S_v^i \cap U$ over all $v \in R^i \setminus \{s\}$ satisfying $w(\partial S_v^i) \leq W$ and $|S_v^i \cap U| \leq z$
5:     $R^{i+1} \leftarrow$ subsample of $R^i$ where each vertex in $R^i \setminus \{s\}$ is sampled independently with probability 1/2, and $s$ is sampled with probability 1
6: return $D^0 \cup D^1 \cup \cdots \cup D^{\lfloor \lg |U| \rfloor}$
```

side containing v has at most z vertices in U.

The lemma below is similar to Lemma 4.5.1 and their proofs share a lot of overlap.

Lemma 4.6.1: D covers a large fraction of D^* in expectation

$D \subseteq D^*$ and $\mathbb{E}[|D|] = \Omega(|D^*|/\log |U|)$.

Proof. We first prove that $D \subseteq D^*$. Each vertex $u \in D$ belongs to some S_v^i satisfying $w(\partial S_v^i) \leq W$ and $|S_v^i \cap U| \leq z$. In particular, ∂S_v^i is an (s, u)-cut with weight at most W whose side S_v^i containing u has at most z vertices in U, so $u \in D^*$.

It remains to prove that $\mathbb{E}[|D|] \geq \Omega(|D^*|/\log |U|)$. Consider a rooted minimal Steiner Gomory-Hu tree T of G on terminals U rooted at s, which exists by Theorem 4.4.4. For each vertex $v \in U \setminus \{s\}$, let $r(v)$ be defined as the child vertex of the lowest weight edge on the path from v to s in T. If there are multiple lowest weight edges, choose the one with the maximum depth.

For each vertex $v \in D^*$, consider the subtree rooted at v, define $U_v \subseteq D^*$ to be the vertices in the subtree, and define n_v as the number of vertices in the subtree. We say that a vertex $v \in D^*$ is *active* if $v \in R^i$ for $i = \lfloor \lg n_{r(v)} \rfloor$. In addition, if $U_{r(v)} \cap R^i = \{v\}$, then we say that v *hits* all of the vertices in $U_{r(v)}$ (including itself); see Figure 4.1 again. In particular, in order for v to hit any other vertex, it must be active. For completeness, we say that any vertex in $U \setminus D^*$ is not active and does not hit any vertex.

To prove that $\mathbb{E}[|D|] \geq \Omega(|D^*|/\log |U|)$, we will show that

(a) each vertex u that is hit is in D,

(b) the total number of pairs (u, v) for which $v \in D^*$ hits u is at least $c|D^*|$ in expectation for some small enough constant $c > 0$, and

(c) with probability at least $1 - \frac{c}{2|U|^2}$ (for the constant $c > 0$ in (b)), each vertex u is hit by at most $O(\log |U|)$ vertices $v \in D^*$.

For (a), consider the path from u to the root s in T, and take any vertex $v \in D^*$ on the path that is active (possibly u itself). Such a vertex must exist since u is hit by some vertex. By definition, for $i = \lfloor \lg n_{r(v)} \rfloor$, we have $U_{r(v)} \cap R^i = \{v\}$, so $\partial f^{-1}(U_{r(v)})$ is a $(v, R^i \setminus \{v\})$-cut. By the definition of $r(v)$, we have that $\partial f^{-1}(U_{r(v)})$ is a (v, s)-mincut. On the other hand, we have that ∂S_v^i is a $(v, R^i \setminus \{v\})$-mincut, so in particular, it is a (v, s)-cut. It follows that $\partial f^{-1}(U_{r(v)})$ and ∂S_v^i are both (v, s)-mincuts and $(v, R^i \setminus v)$-mincuts, and $w(\partial S_v^i) = \mathsf{mincut}(s, v) \le W$. Since T is a minimal Gomory-Hu Steiner tree, we must have $f^{-1}(U_{r(v)}) \subseteq S_v^i$. Since S_v^i is the minimal $(v, R^i \setminus \{v\})$-mincut, it is also the minimal (v, s)-mincut, so $S_v^i \subseteq f^{-1}(U_{r(v)})$. It follows that $f^{-1}(U_{r(v)}) = S_v^i$. Since $f^{-1}(U_{r(v)})$ is the minimal (v, s)-mincut and $v \in D^*$, we must have $|f^{-1}(U_{r(v)}) \cap U| \le z$, so in particular, $|S_v^i \cap U| = |f^{-1}(U_{r(v)}) \cap U| \le z$. Therefore, the vertex v satisfies all the conditions of line 2c. Moreover, since $u \in U_{r(v)} \subseteq f^{-1}(U_{r(v)}) = S_v^i$, vertex u is added to D in the set $S_v^i \cap U$.

For (b), for $i = \lfloor \lg n_{r(v)} \rfloor$, we have $v \in R^i$ with probability exactly $1/2^i = \Theta(1/n_{r(v)})$, and with probability $\Omega(1)$, no other vertex in $U_{r(v)}$ joins R^i. Therefore, v is active with probability $\Omega(1/n_{r(v)})$. Conditioned on v being active, it hits exactly $n_{r(v)}$ many vertices. It follows that v hits $\Omega(1)$ vertices in expectation.

For (c), the number of vertices v that hit vertex u is at most the number of active vertices v for which $r(v)$ is on the path from u to s in T. Label these vertices $u = v_1, v_2, \ldots, v_\ell = s$, ordered by increasing distance from u to $r(v_i)$ in T. Each vertex $v_j \in D^*$ is active with probability $\Theta(1/n_{r(v_j)})$, which is at most $\Theta(1/j)$ since $v_1, \ldots, v_j \in U_{r(v_j)}$. Each vertex $v_j \notin D^*$ is never active. Therefore, the expected number of active vertices on the path from u to s is at most $\sum_{j=1}^{\ell} \Theta(1/j) = \Theta(\ln \ell) \le \Theta(\ln |U|)$. A standard Chernoff bound shows that with probability at least $1 - \frac{c}{2|U|^3}$ for any constant $c > 0$, the number of active vertices on the path is indeed $O(\ln |U|)$, where the $O(\cdot)$ hides the dependency on c. Taking a union bound over all $u \in U$, the probability that this is true for all vertices is at least $1 - \frac{c}{2|U|^2}$.

Finally, we show why properties (a) to (c) imply $\mathbb{E}[|D|] \ge \Omega(|D^*|/\log |U|)$. In the event that property (c) fails, the total number of pairs (u, v) for which v hits u can be trivially upper bounded by $|U|^2$. Since this occurs with probability at most $\frac{c}{2|U|^2}$, the total contribution to the expectation $c|D^*|$ in property (b) is at most $c/2$. Therefore, the contribution to the expectation in the event that property (c) succeeds is at least $c|D^*| - c/2 \ge (c/2)|D^*|$. In this case, since each vertex is hit at most $O(\log |U|)$ times, there are at least $\Omega(|D^*|/\log |U|)$ vertices hit in expectation, all of which are included in D by property (a). □

We now use iterate Algorithm CutThresholdStep to obtain the CutThreshold algorithm:

Algorithm 6 CutThreshold$(G = (V, E), s, W)$

```
1: Initialize $U \leftarrow V$ and $D_{\text{total}} \leftarrow \emptyset$
2: for $O(\log^2 n)$ iterations do
3:     Let $D$ be the union of the sets output by CutThresholdStep$(G, s, U, W, \infty)$
4:     Update $D_{\text{total}} \leftarrow D_{\text{total}} \cup D$ and $U \leftarrow U \setminus D$
5: return $D_{\text{total}}$
```

Corollary 4.6.2: Correctness of CutThreshold

W.h.p., the output D_{total} of CutThreshold is exactly all vertices $v \in U \setminus \{s\}$ for which the (s, v)-mincut has weight at most W.

Proof. By Lemma 4.6.1, $|U \cap D^*|$ decreases by $\Omega(|D^*|/\log n)$ in expectation. After $O(\log^2 n)$ iterations, we have $\mathbb{E}[|U \cap D^*|] \leq 1/\text{poly}(n)$, so w.h.p., $U \cap D^* = \emptyset$. Each vertex in D^* that is removed from U is added to D_{total}, and no vertices in $U \setminus D^*$ are added to D_{total}, so w.h.p., the algorithm returns the correct set D^*. □

In other words, CutThreshold is an algorithm that fulfills Theorem 4.2.8.

4.7 Approximate GH Tree

Let $\epsilon > 0$ be a fixed parameter throughout the recursive algorithm. We present our approximate Steiner Gomory-Hu tree algorithm in ApproxSteinerGHTree below. See Figure 4.2 for a visual guide to the algorithm.

At a high level, the algorithm applies divide-and-conquer by cutting the graph along sets S_v^i computed by CutThresholdStep, applying recursion to each piece, and stitching the recursive Gomory-Hu trees together in the same way as the standard recursive Gomory-Hu tree construction. To avoid complications, we only select sets S_v^i from a *single* level $i \in \{0, 1, 2 \ldots, \lfloor \lg |U| \rfloor\}$, which are guaranteed to be vertex-disjoint. Furthermore, instead of selecting all sets $\{S_v^i : v \in R^i\}$, we only select those for which $|S_v^i \cap U| \leq |U|/2$; this allows us to bound the recursion depth. By choosing the source $s \in U$ at *random*, we guarantee that in expectation, we do not exclude too many sets S_v^i. The chosen sets partition the graph into disjoint sets of vertices (including the set of vertices outside of any chosen set S_v^i). We split the graph along this partition a similar way to the standard Gomory-Hu tree construction: for each set in the partition, contract all other vertices into a single vertex and recursively compute the Steiner Gomory-Hu tree of the contracted graph. This gives us a collection of Gomory-Hu Steiner trees, which we then stitch together into a single Gomory-Hu Steiner tree in the standard way.

4.7.1 Approximation

Since the approximation factors can potentially add up down the recursion tree, we need to bound the depth of the recursive algorithm. Here, there are two types of recursion: the recursive calls (G_v, U_v), and the single call $(G_{\text{large}}, U_{\text{large}})$. Taking a branch down (G_v, U_v) is easy: since $|U_v| \leq |U|/2$, the algorithm can travel down such a branch at most $\lg |U|$ times. The difficult part is in bounding the number of branches down $(G_{\text{large}}, U_{\text{large}})$. It turns out that after polylog(n) consecutive branches down $(G_{\text{large}}, U_{\text{large}})$, the Steiner mincut increases by factor $(1 + \epsilon)$, w.h.p.; we elaborate on this insight in Section 4.7.2, which concerns the

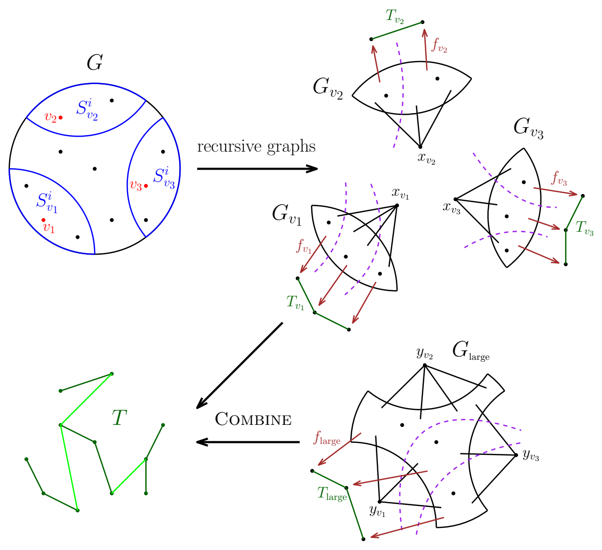

Figure 4.2: Recursive construction of G_{large} and G_v for $v \in R^i_{\text{small}}$. Here, $R^i_{\text{small}} = \{v_1, v_2, v_3\}$, denoted by red vertices on the top left. The dotted blue curves on the right mark the boundaries of the regions $f^{-1}_{v_i}(u) : u \in U_{v_i}$ and $f^{-1}_{v_{\text{large}}}(u) : u \in U_{\text{large}}$. The light green edges on the bottom left are the edges $(f_{v_i}(x_{v_i}), f_{\text{large}}(y_{v_i}))$ added on line 1 of COMBINE.

Algorithm 7 APPROXSTEINERGHTREE$(G = (V, E), U)$

```
1: $\lambda \leftarrow$ global Steiner mincut of $G$ with terminals $U$
2: $s \leftarrow$ uniformly random vertex in $U$
3: Call CUTTHRESHOLDSTEP$(G, s, U, (1+\epsilon)\lambda, |U|/2)$, and let $R^j$ and $S^j_v : v \in R^j$ $(0 \le j \le$
   $\lg |U|)$ be the intermediate variables in the algorithm
4: Let $i \in \{0, 1, \dots, \lfloor \lg |U| \rfloor\}$ be the iteration maximizing $\left|\bigcup_{v \in R^i} (S^i_v \cap U)\right|$

5: for each $v \in R^i$ do                 ▷ Construct recursive graphs and apply recursion
6:     Let $G_v$ be the graph $G$ with vertices $V \setminus S^i_v$ contracted to a single vertex $x_v$ ▷ $S^i_v$ are
   disjoint
7:     Let $U_v \leftarrow S^i_v \cap U$
8:     $(T_v, f_v) \leftarrow$ APPROXSTEINERGHTREE$(G_v, U_v)$
9: Let $G_{\text{large}}$ be the graph $G$ with (disjoint) vertex sets $S^i_v$ contracted to single vertices $y_v$
   for all $v \in R^i$
10: Let $U_{\text{large}} \leftarrow U \setminus \bigcup_{v \in R^i} (S^i_v \cap U)$
11: $(T_{\text{large}}, f_{\text{large}}) \leftarrow$ APPROXSTEINERGHTREE$(G_{\text{large}}, U_{\text{large}})$

12: Combine $(T_{\text{large}}, f_{\text{large}})$ and $\{(T_v, f_v) : v \in R^i\}$ into $(T, f)$ according to COMBINE
13: return $(T, f)$
```

running time. Since the Steiner mincut can never decrease down any recursive branch, it can increase by factor $(1+\epsilon)$ at most $\epsilon^{-1}\text{polylog}(n)\log\Delta$ times. Thus, we have a bound of $\epsilon^{-1}\text{polylog}(n)\log\Delta$ on the recursion depth, w.h.p.

This depth bound alone is not enough for the following reason: if the approximation factor increase by $(1+\epsilon)$ along each recursive branch, then the total approximation becomes $(1+\epsilon)^{\epsilon^{-1}\text{polylog}(n)\log\Delta}$, which is no good because the $(1+\epsilon)$ and ϵ^{-1} cancel each other. Here, our key insight is that actually, the approximation factor does not distort *at all* down $(G_{\text{large}}, U_{\text{large}})$. It may increase by factor $(1+\epsilon)$ down any (G_v, U_v), but this can only happen $\lg|U|$ times, giving us an approximation factor of $(1+\epsilon)^{\lg|U|}$, which is fine because we can always retroactively replace ϵ with $\Theta(\epsilon/\lg|U|)$ to obtain the desired $(1+\epsilon)$.

The lemma below formalizes our insight that approximation factors are preserved down the branch $(G_{\text{large}}, U_{\text{large}})$.

Lemma 4.7.1: Mincuts in G_{large} are preserved exactly

For any distinct vertices $p, q \in U_{\text{large}}$, we have $\mathsf{mincut}_{G_{\text{large}}}(p,q) = \mathsf{mincut}_G(p,q)$.

Proof. Since G_{large} is a contraction of G, we have $\mathsf{mincut}_{G_{\text{large}}}(p,q) \ge \mathsf{mincut}_G(p,q)$. To show the reverse inequality, fix any (p,q)-mincut in G, and let S be one side of the mincut. We show that for each $v \in R^i$, either $S_v^i \subseteq S$ or $S_v^i \subseteq V \setminus S$. Assuming this, the cut ∂S stays intact when the sets S_v^i are contracted to form G_{large}, so $\mathsf{mincut}_{G_{\text{large}}}(p,q) \le w(\partial S) = \mathsf{mincut}_G(p,q)$.

Consider any $v \in R^i$, and suppose first that $v \in S$. Then, $S_v^i \cap S$ is still a $(v, R^i \setminus v)$-cut, and $S_v^i \cup S$ is still a (p,q)-cut. By the submodularity of cuts,

$$w(\partial_G S_v^i) + w(\partial_G S) \ge w(\partial_G(S_v^i \cup S)) + w(\partial_G(S_v^i \cap S)).$$

In particular, $S_v^i \cap S$ must be a minimum $(v, R^i \setminus v)$-cut. Since S_v^i is the minimal $(v, R^i \setminus v)$-mincut, it follows that $S_v^i \cap S = S_v^i$, or equivalently, $S_v^i \subseteq S$.

Suppose now that $v \notin S$. In this case, we can swap p and q, and swap S and $V \setminus S$, and repeat the above argument to get $S_v^i \subseteq V \setminus S$. □

Similarly, the lemma below says that approximation factors distort by at most $(1+\epsilon)$ down a (G_v, U_v) branch.

Lemma 4.7.2: Mincuts in G_v are preserved $(1+\epsilon)$-approximately

For any $v \in R^i$ and any distinct vertices $p, q \in U_v$, we have $\mathsf{mincut}_G(p,q) \le \mathsf{mincut}_{G_v}(p,q) \le (1+\epsilon)\mathsf{mincut}_G(p,q)$.

Proof. The lower bound $\mathsf{mincut}_G(p,q) \le \mathsf{mincut}_{G_v}(p,q)$ holds because G_v is a contraction of G, so we focus on the upper bound. Fix any (p,q)-mincut in G, and let S be the side of the mincut not containing s (recall that $s \in U$ and $s \notin S_v^i$). Since $S_v^i \cup S$ is a (p,s)-cut (it is also a (q,s)-cut), it is in particular a Steiner cut for terminals U, so $w(S_v^i \cup S) \geq \lambda$. Also, $w(S_v^i) \leq (1+\epsilon)\lambda$ by the choice of the threshold $(1+\epsilon)\lambda$ (line 3). Together with the submodularity of cuts, we obtain

$$\begin{aligned}(1+\epsilon)\lambda + w(\partial_G S) &\geq w(\partial_G S_v^i) + w(\partial_G S) \\ &\geq w(\partial_G(S_v^i \cup S)) + w(\partial_G(S_v^i \cap S)) \\ &\geq \lambda + w(\partial_G(S_v^i \cap S)).\end{aligned}$$

The set $S_v^i \cap S$ stays intact under the contraction from G to G_v, so $w(\partial_{G_v}(S_v^i \cap S)) = w(\partial_G(S_v^i \cap S))$. Therefore,

$$\begin{aligned}\mathsf{mincut}_{G_v}(p,q) &\leq w(\partial_{G_v}(S_v^i \cap S)) \\ &= w(\partial_G(S_v^i \cap S)) \\ &\leq w(\partial_G S) + \epsilon\lambda \\ &\leq \mathsf{mincut}_G(p,q) + \epsilon\,\mathsf{mincut}_G(p,q),\end{aligned}$$

as promised. □

Finally, the lemma below determines our final approximation factor.

Lemma 4.7.3: Approximation factor

ApproxSteinerGHTree$(G=(V,E),U)$ outputs a $(1+\epsilon)^{\lg |U|}$-approximate Gomory-Hu Steiner tree.

Proof. We apply induction on $|U|$. Since $|U_v| \leq |U|/2$ for all $v \in R^i$, by induction, the recursive outputs (T_v, f_v) are Gomory-Hu Steiner trees with approximation $(1+\epsilon)^{\lg|U_v|} \leq (1+\epsilon)^{\lg|U|-1}$. By definition, this means that for all $s,t \in U_v$ and the minimum-weight edge (u,u') on the s–t path in T_v, letting $U_v' \subseteq U_v$ be the vertices of the connected component of $T_v - (u,u')$ containing s, we have that $f_v^{-1}(U_v')$ is a $(1+\epsilon)^{\lg|U|-1}$-approximate (s,t)-mincut in G_v with value is $w_T(u,u')$. Define $U' \subseteq U$ as the vertices of the connected component of $T-(u,u')$ containing s. By construction of (T,f) (lines 1 and 2), the set $f^{-1}(U')$ is simply $f_v^{-1}(U_v')$ with the vertex x_v replaced by $V \setminus S_v^i$ in the case that $x_v \in f^{-1}(U')$. Since G_v is simply G with all vertices $V\setminus S_v^i$ contracted to x_v, we conclude that $w_{G_v}(\partial f_v^{-1}(U_v')) = w_G(\partial f^{-1}(U'))$. By Lemma 4.7.2, the values $\mathsf{mincut}_G(s,t)$ and $\mathsf{mincut}_{G_v}(s,t)$ are within factor $(1+\epsilon)$ of each other, so $w_G(\partial f^{-1}(U'))$ approximates the (s,t)-mincut in G to a factor $(1+\epsilon)\cdot(1+\epsilon)^{\lg|U|-1} = (1+\epsilon)^{\lg|U|}$. In other words, the Gomory-Hu Steiner tree condition for (T,f) is satisfied for all $s,t \in U_v$ for some $v \in R^i$.

By induction, the recursive output $(T_{\text{large}}, f_{\text{large}})$ is a Gomory-Hu Steiner tree with approximation $(1+\epsilon)^{\lg|U_{\text{large}}|} \leq (1+\epsilon)^{\lg|U|}$. Again, consider $s,t \in U_{\text{large}}$ and the minimum-weight edge (u,u') on the s–t path in T_{large}, and let $U'_{\text{large}} \subseteq U_{\text{large}}$ be the vertices of the connected component of $T_{\text{large}} - (u,u')$ containing s. Define $U' \subseteq U$ as the vertices of the connected component of $T - (u, u')$ containing s. By a similar argument, we have $w_{G_{\text{large}}}(\partial f^{-1}_{\text{large}}(U'_{\text{large}})) = w_G(\partial f^{-1}(U'))$. By Lemma 4.7.1, we also have $\mathsf{mincut}_G(s,t) = \mathsf{mincut}_{G_{\text{large}}}(s,t)$, so $w_G(\partial f^{-1}(U'))$ is a $(1+\epsilon)^{\lg |U|}$-approximate (s,t)-mincut in G, fulfilling the Gomory-Hu Steiner tree condition for (T, f) in the case $s, t \in U_{\text{large}}$.

There are two remaining cases: $s \in U_v$ and $t \in U_{v'}$ for distinct $v, v' \in R^i$, and $s \in U_v$ and $t \in U_{\text{large}}$; we treat both cases simultaneously. Since G has Steiner mincut λ, each of the contracted graphs G_{large} and G_v has Steiner mincut at least λ. By induction, every edge in T_v and T_{large} or $T_{v'}$ (depending on case) has weight at least $(1+\epsilon)^{-\lg|U|}\lambda$. By construction, the s–t path in T has at least one edge of the form $(f_v(x_v), f_{\text{large}}(y_v))$, added on line 1; this edge has weight $w(\partial_G S_v^i) \le (1+\epsilon)\lambda$. Therefore, the minimum-weight edge on the s–t path in T has weight at least $(1+\epsilon)^{-\lg|U|}\lambda$ and at most $(1+\epsilon)\lambda$; in particular, it is a $(1+\epsilon)^{\lg|U|}$-approximation of $\mathsf{mincut}_G(s,t)$. If the edge is of the form $(f_v(x_v), f_{\text{large}}(y_v))$, then by construction, the relevant set $f^{-1}(U')$ is exactly S_v^i, which is a $(1+\epsilon)$-approximate (s,t)-mincut in G. If the edge is in T_{large} or T_v or $T_{v'}$, then we can apply the same arguments used previously. □

4.7.2 Running Time Bound

In order for a recursive algorithm to be efficient, it must make substantial progress on each of its recursive calls, which can then be used to bound its depth. For each recursive call (G_v, U_v, ϵ), we have $|U_v| \le |U|/2$ by construction, so we can set our measure of progress to be $|U|$, the number of terminals, which halves upon each recursive call. However, progress on $(G_{\text{large}}, U_{\text{large}}, \epsilon)$ is unclear; in particular, it is possible for $|U_{\text{large}}|$ to be very close to $|U|$ with probability 1. For G_{large}, we define the following alternative measure of progress. Let $P(G, U, W)$ be the set of unordered pairs of distinct vertices whose mincut is at most W:

$$P(G,U,W) = \left\{ \{u,v\} \in \binom{U}{2} : \mathsf{mincut}_G(u,v) \le W \right\}.$$

In particular, we will consider its size $|P(G,U,W)|$, and show the following expected reduction:

Lemma 4.7.4: Expected reduction of $|P(G,U,W)|$ in $(G_{\text{large}}, U_{\text{large}})$

For any $W \le (1+\epsilon)\lambda$, over the random selection of s and the randomness in CutThresholdStep, we have

$$\mathbb{E}[|P(G_{\text{large}}, U_{\text{large}}, W)|] \le \left(1 - \Omega\left(\frac{1}{\log^2 n}\right)\right) |P(G,U,W)|.$$

Before we prove Lemma 4.7.4, we show how it implies progress on the recursive call for G_{large}.

Corollary 4.7.5:

Let λ_0 be the global Steiner mincut of G. W.h.p., after $\Omega(\log^3 n)$ recursive calls along G_{large} (replacing $G \leftarrow G_{\text{large}}$ each time), the global Steiner mincut of G is at least $(1+\epsilon)\lambda_0$ (where λ_0 is still the global Steiner mincut of the initial graph).

Proof. Let $W = (1+\epsilon)\lambda_0$. Initially, we trivially have $|P(G,U,W)| \le \binom{|U|}{2}$. The global Steiner mincut can only increase in the recursive calls, since G_{large} is always a contraction of G, so we always have $W \le (1+\epsilon)\lambda$ for the current global Steiner mincut λ. By Lemma 4.7.4, the value $|P(G,U,W)|$ drops by factor $1-\Omega(\frac{1}{\log^2 n})$ in expectation on each recursive call, so after $\Omega(\log^3 n)$ calls, we have

$$\mathbb{E}[|P(G,U,W)|] \le \binom{|U|}{2} \cdot \left(1 - \Omega\left(\frac{1}{\log^2 n}\right)\right)^{\Omega(\log^3 n)} \le \frac{1}{\text{poly}(n)}.$$

In other words, w.h.p., we have $|P(G,U,W)| = 0$ at the end, or equivalently, the Steiner mincut of G is at least $(1+\epsilon)\lambda$. □

Combining both recursive measures of progress together, we obtain the following bound on the recursion depth:

Lemma 4.7.6: Recursion depth bound of ApproxSteinerGHTree

Let $w_{\min}$ and $w_{\max}$ be the minimum weight and maximum weight of any edge in G. W.h.p., the depth of the recursion tree of ApproxSteinerGHTree is $O(\epsilon^{-1}\log^3 n \log(n\Delta))$.

Proof. For any $\Theta(\log^3 n)$ successive recursive calls down the recursion tree, either one call was on a graph G_v, or $\Theta(\log^3 n)$ of them were on the graph G_{large}. In the former case, $|U|$ drops by half, so it can happen $O(\log n)$ times total. In the latter case, by Corollary 4.7.5, the global Steiner mincut increases by factor $(1+\epsilon)$. Let $w_{\min}$ and $w_{\max}$ be the minimum and maximum weights in G, so that $\Delta = w_{\max}/w_{\min}$. Note that for any recursive instance (G',U') and any $s,t \in U'$, we have $w_{\min} \le \mathsf{mincut}_{G'}(s,t) \le w(\partial(\{s\})) \le nw_{\max}$, so the global Steiner mincut of (G',U') is always in the range $[w_{\min}, nw_{\max}]$. It follows that calling G_{large} can happen $O(\epsilon^{-1}\log(nw_{\max}/w_{\min}))$ times, hence the bound. □

We state the next theorem for *unweighted* graphs only. For weighted graphs, there is no nice bound on the number of new edges created throughout the algorithm, and therefore no easy bound on the overall running time. In the next section, we introduce a graph sparsification step to handle this issue.

Lemma 4.7.7: Running time bound of ApproxSteinerGHTree, unweighted graphs only

For an *unweighted* graph $G = (V, E)$, and terminals $U \subseteq V$, ApproxSteinerGHTree(G, V, ϵ) takes time $\tilde{O}(m\epsilon^{-1})$ plus calls to max-flow on instances with a total of $\tilde{O}(n\epsilon^{-1})$ vertices and $\tilde{O}(m\epsilon^{-1})$ edges.

Proof. For a given recursion level, consider the instances $\{(G_i, U_i, W_i)\}$ across that level. By construction, the terminals U_i partition U. Moreover, the total number of vertices over all G_i is at most $n + 2(|U| - 1) = O(n)$ since each branch creates 2 new vertices and there are at most $|U| - 1$ branches. The total number of new edges created is at most the sum of weights of the edges in the final $(1 + \epsilon)$-approximate Gomory-Hu Steiner tree. For an unweighted graph, this is $O(m)$ by the following well-known argument. Root the Gomory-Hu Steiner tree T at any vertex $r \in U$; for any $v \in U \setminus r$ with parent u, the cut $\partial\{v\}$ in G is a (u, v)-cut of value $\deg(v)$, so $w_T(u, v) \le \deg(v)$. Overall, the sum of the edge weights in T is at most $\sum_{v\in U} \deg(v) \le 2m$.

Therefore, there are $O(n)$ vertices and $O(m)$ edges in each recursion level. By Lemma 4.7.6, there are $O(\epsilon^{-1} \log^4 n)$ levels (since $\Delta = 1$ for an unweighted graph), for a total of $\tilde{O}(n\epsilon^{-1})$ vertices and $\tilde{O}(m\epsilon^{-1})$ edges. In particular, the instances to the max-flow calls have $\tilde{O}(n\epsilon^{-1})$ vertices and $\tilde{O}(m\epsilon^{-1})$ edges in total. ☐

Combining Lemmas 4.7.3 and 4.7.7 and resetting $\epsilon \leftarrow \Theta(\epsilon/ \log n)$, we obtain Theorem 4.2.6.

Finally, we prove Lemma 4.7.4.

Proof (Lemma 4.7.4). Let D^* be all vertices $v \in U \setminus s$ for which there exists an (s, v)-cut of weight at most W whose side containing v has at most $|U|/2$ vertices in U. Define $D = \bigcup_{j=0}^{\lfloor \lg |U| \rfloor} \bigcup_{v\in R^i} (S_v^i \cap U)$. Let $P_{\text{ordered}}(G, U, W)$ be the set of ordered pairs $(u, v) : u, v \in V$ for which there exists an (u, v)-mincut of weight at most W with at most $|U|/2$ vertices in U on the side $S(u, v) \subseteq V$ containing u. We now state and prove the following four properties:

(a) For all $u, v \in U$, $\{u, v\} \in P(G, U, W)$ if and only if either $(u, v) \in P_{\text{ordered}}(G, U, W)$ or $(v, u) \in P_{\text{ordered}}(G, U, W)$ (or both).

(b) For each pair $(u, v) \in P_{\text{ordered}}(G, U, W)$, we have $u \in D^*$ with probability at least 1/2,

(c) For each $u \in D^*$, there are at least $|U|/2$ vertices $v \in U$ for which $(u, v) \in P_{\text{ordered}}(G, U, W)$.

(d) Over the randomness in CutThresholdStep$(G, U, (1 + \epsilon)\lambda)$, $\mathbb{E}[|D|] \ge \Omega(|D^*|/ \log |U|)$.

Property (a) follows by definition. Property (b) follows from the fact that $u \in D^*$ whenever $s \notin S(u, v)$, which happens with probability at least 1/2. Property (c) follows because any vertex $v \in U \setminus S(u, v)$ satisfies $(u, v) \in P_{\text{ordered}}(G, U, W)$, of which there are at least $|U|/2$. Property (d) follows from Lemma 4.6.1 applied on CutThresholdStep$(G, U, W, |U|/2)$, and then observing that even though we actually call CutThresholdStep$(G, U, (1 +$

$\epsilon)\lambda, |U|/2)$, the set D can only get larger if the weight parameter is increased from W to $(1+\epsilon)\lambda$.

With properties (a) to (d) in hand, we now finish the proof of Lemma 4.7.4. Consider the iteration i maximizing the size of $D^i := \bigcup_{v\in R^i}(S_v^i \cap U)$ (line 4), so that $|D^i| \geq |D|/(\lfloor \lg |U| \rfloor + 1)$. For any vertex $u \in D^i$, all pairs $(u,v) \in P_{\text{ordered}}(G,U,W)$ (over all $v \in U$) disappear from $P_{\text{ordered}}(G,U,W)$, which is at least $|U|/2$ many by (c). In other words,

$$\begin{aligned}&|P_{\text{ordered}}(G,U,W) \setminus P_{\text{ordered}}(G_{\text{large}},U_{\text{large}},W)|\\ &\geq \frac{|U|}{2}|D^i|\\ &\geq \Omega\left(\frac{|U|\cdot|D|}{\log|U|}\right).\end{aligned}$$

Taking expectations and applying (d),

$$\begin{aligned}&\mathbb{E}[|P_{\text{ordered}}(G,U,W) \setminus P_{\text{ordered}}(G_{\text{large}},U_{\text{large}},W)|]\\ &\geq \Omega\left(\frac{|U|\cdot\mathbb{E}[|D|]}{\log|U|}\right)\\ &\geq \Omega\left(\frac{|U|\cdot|D^*|}{\log^2|U|}\right).\end{aligned}$$

Moreover,

$$|U|\cdot|D^*| \geq \mathbb{E}\big[\big|\{(u,v) : u \in D^*\}\big|\big] \geq \frac{1}{2}|P_{\text{ordered}}(G,U,W)|,$$

where the second inequality follows by (b). Putting everything together, we obtain

$$\begin{aligned}&\mathbb{E}[|P_{\text{ordered}}(G,U,W) \setminus P_{\text{ordered}}(G_{\text{large}},U_{\text{large}},W)|]\\ &\geq \Omega\left(\frac{|P_{\text{ordered}}(G,U,W)|}{\log|U|}\right).\end{aligned}$$

Finally, applying (a) gives

$$\mathbb{E}[|P(G,U,W) \setminus P(G_{\text{large}},U_{\text{large}},W)|] \geq \Omega\left(\frac{|P(G,U,W)|}{\log|U|}\right).$$

Finally, we have $P(G_{\text{large}},U_{\text{large}},W) \subseteq P(G,U,W)$ since the (u,v)-mincut for $u,v \in U_{\text{large}}$ can only increase in G_{large} due to G_{large} being a contraction of G (in fact it says the same by Lemma 4.7.1). Therefore,

$$\begin{aligned}&|P(G,U,W)| - |P(G_{\text{large}},U_{\text{large}},W)|\\ &= |P(G,U,W) \setminus P(G_{\text{large}},U_{\text{large}},W)|,\end{aligned}$$

and combining with the bound on $\mathbb{E}[|P(G,U,W) \setminus P(G_{\text{large}},U_{\text{large}},W)|]$ concludes the proof. $\square$

4.7.3 Weighted Graphs

For weighted graphs, we cannot easily bound the total size of the recursive instances. Instead, to keep the sizes of the instances small, we sparsify the recursive instances to have roughly the same number of edges and vertices. By the proof of Lemma 4.7.7, the total number of vertices over all instances in a given recursion level is at most $n + 2(|U| - 1) = O(n)$. Therefore, if each such instance is sparsified, the total number of edges becomes $\tilde{O}(n)$, and the algorithm is efficient.

It turns out we only need to re-sparsify the graph in two cases: when we branch down to a graph G_v (and not G_{large}), and when the mincut λ increases by a constant factor, say 2. The former can happen at most $O(\log n)$ times down any recursion branch, since $|U|$ decreases by a factor 2 each time, and the latter occurs $O(\log(n\Delta))$ times down any branch. Each time, we sparsify up to factor $1 + \Theta(\epsilon/\log(n\Delta))$, so that the total error along any branch is $1 + \Theta(\epsilon)$.

We now formalize our arguments. We begin with the specification routine due to Benczur and Karger [15].

Theorem 4.7.8: Graph sparsification

Given a weighted, undirected graph G, and parameters $\epsilon, \delta > 0$, there is a randomized algorithm that with probability at least $1 - \delta$ outputs a $(1+\epsilon)$-approximate sparsifier of G with $O(n\epsilon^{-2}\log(n/\delta))$ edges.

We now derive approximation and running time bounds.

Theorem 4.7.9: Running time bound of ApproxSteinerGHTree, weighted graphs

Suppose that the recursive algorithm ApproxSteinerGHTree sparsifies the input in the following three cases, using Theorem 4.7.8 with the same parameter ϵ and the parameter $\delta = 1/\text{poly}(n)$:

1. The instance was the original input, or
2. The instance was obtained from calling (G_v, U_v), or
3. The instance was obtained from calling $(G_{\text{large}}, U_{\text{large}})$, and the Steiner mincut increased by a factor of at least 2 since the last sparsification.

Then w.h.p., the algorithm outputs a $(1+\epsilon)^{O(\log(n\Delta))}$-approximate Gomory-Hu Steiner tree and takes $\tilde{O}(m)$ time plus calls to max-flow on instances with a total of $\tilde{O}(n\epsilon^{-1}\log\Delta)$ vertices and $\tilde{O}(n\epsilon^{-1}\log\Delta)$ edges.

Proof. We first argue about the approximation factor. Along any branch of the recursion tree, there is at most one sparsification step of type (1), at most $O(\log n)$ sparsification steps of type (2), and at most $O(\log(n\Delta))$ sparsification steps of type (3). Each sparsification distorts the pairwise mincuts by a $(1+\epsilon)$ factor, so the total distortion is $(1+\epsilon)^{O(\log(n\Delta))}$.

Next, we consider the running time. The recursion tree can be broken into chains of recursive G_{large} calls, so that each chain begins with either the original instance or some intermediate G_v call, which is sparsified by either (1) or (2). Fix a chain, and let n' be the number of vertices at the start of the chain, so that the number of edges is $O(n' \log n)$. Within each chain, the number of vertices can only decrease down the chain. After each sparsification, many sparsifications of type (2), and between two consecutive sparsifications, the number of edges can only decrease down the chain since the graph can only contract. It follows that each instance in the chain has at most n' vertices and $O(n'\epsilon^{-2}\log n)$ edges. By Lemma 4.7.6, each chain has length $O(\epsilon^{-1}\log^3 n \log(n\Delta))$, so the total number of vertices and edges in the chain is $\tilde{O}(n'\epsilon^{-3}\log\Delta)$. Imagine charging these vertices and edges to the n' vertices at the root of the chain. In other words, to bound the total number of edges in the recursion tree, it suffices to bound the total number of vertices in the original instance and in intermediate G_v calls.

In the recursion tree, there are n original vertices and at most $2(|U|-1)$ new vertices, since each branch creates 2 new vertices and there are at most $|U|-1$ branches. Each vertex joins $O(\log n)$ many G_v calls, since every time a vertex joins one, the number of terminals drops by half; note that a vertex is never duplicated in the recursion tree. It follows that there are $O(n \log n)$ many vertices in intermediate G_v calls, along with the n vertices in the original instance. Hence, from our charging scheme, we conclude that there are a total of $\tilde{O}(n\epsilon^{-3}\log\Delta)$ vertices and edges in the recursion tree. In particular, the instances to the max-flow calls have $\tilde{O}(n\epsilon^{-3}\log\Delta)$ vertices and edges in total. □

Resetting $\epsilon \leftarrow \Theta(\epsilon/\log(n\Delta))$, we have thus proved Theorem 4.2.5.

4.8 Conclusion

In this chapter, we presented a reduction from exact Gomory-Hu tree to a simpler, seemingly more tractable problem which we named single-source mincut verification. Although we could not improve the running time of the latter beyond the trivial $n-1$ max-flows, we showed that improvements could be made in the approximate version. Our main algorithmic result was an $(1+\epsilon)$-approximate Gomory-Hu tree in roughly (exact) max-flow time.

Of course, the reduction from approximate Gomory-Hu tree to exact max-flow is unsatisfactory; a reduction to approximate max-flow would instead lead to a near-linear time algorithm. However, there is a fundamental barrier to this endeavor: the isolating cuts lemma (as stated in Lemma 2.2.2) does not hold in the approximate setting. Fortunately, we discovered a variant of the isolating cuts lemma that is robust to approximations, although it requires more than just computing approximate mincuts. The additional property we require is that the mincuts are "well-linked" in a sense, and achieving this guarantee is far from trivial.

Chapter 5

Directed Global Mincut

In this chapter, we present our algorithm for global mincut on *directed* graphs based on the work of [18]. While a near-linear time algorithm for the undirected global mincut problem was known since Karger [55], nothing close to linear has been found for directed graphs. Indeed, Karger's undirected global mincut algorithm breaks down in multiple ways in the directed setting. The most glaring issue is the graph sparsification step, namely that directed graphs are notoriously difficult to sparsify. Nevertheless, in this chapter, we show that under a locality assumption, a partial sparsification of directed graphs is possible, demonstrating again how locality can be used to make seemingly impossible problems tractable.

Once again, our specific locality assumption is that the target solution is unbalanced: one side of the mincut has at most r vertices for some parameter r. In this case, we are able to sparsify the directed graph so that the original mincut is still approximately a mincut in the sparsified graph, although the size of the sparsifier is not perfect and depends on the parameter r. we then follow Karger's approach to his near-linear time algorithm for *un*directed graphs to recover the (exact) directed mincut.

To solve the balanced case, we employ a completely different algorithm this time: we simply sample vertices s, t at random and compute the (s, t)-mincut. We succeed if we sample s, t on the correct sides of the mincut, which happens with higher probability when the mincut is balanced. Finally, we optimize the locality parameter $r \approx \sqrt{n}$, achieving a directed mincut algorithm in $\tilde{O}(\sqrt{n})$ max-flow calls. Using the state-of-the-art max-flow algorithms, we obtain the fastest directed mincut algorithm for both sparse and dense graphs.

5.1 Background

Due to the difficulty of the directed graph setting, only a few of the undirected mincut algorithms generalize to directed graphs. The most notable one is Hao and Orlin's $\tilde{O}(mn)$ time algorithm based on the push-relabel max-flow algorithm. Using a different technique of duality between rooted mincuts and arborescences, Gabow [40] obtained a running time of $\tilde{O}(m\lambda)$ for this problem, where λ is the weight of a mincut (assuming integer weights). This is at least as good as the Hao-Orlin running time for unweighted simple graphs, but can be much worse for weighted graphs. Indeed, prior to our work, the Hao-Orlin bound of $\tilde{O}(mn)$ remained the state of the art for the directed mincut problem on arbitrary weighted graphs.

5.2 Our Techniques

At a high level, our algorithm resembles a directed graph version of Karger's near-linear time mincut algorithm in undirected graphs [55] as discussed in Section 6.3. Recall that Karger's algorithm has three main steps: (a) sparsify the graph by random sampling of edges to reduce the mincut value to $O(\log n)$, (b) use a semi-duality between mincuts and spanning trees to pack $O(\log n)$ edge-disjoint spanning trees in the sparsifier, and (c) find the minimum weight cut among those that have only one or two edges in each such spanning tree using a dynamic program. But, directed graphs are substantially different from undirected graphs. In particular, steps (a) and (c) are not valid in a directed graph. We cannot hope to sparsify a directed graph since many directed graphs do not have sparsifiers even in an existential sense. Moreover, even if a mincut had just a single edge in a spanning tree, Karger's dynamic program to recover this cut cannot be used in a directed graph.

To overcome these challenges, we adopt several ingredients that we outline below:

- Inspired by locality, we consider two possibilities: either the mincut has $\tilde{O}(\sqrt{n})$ vertices on the smaller side or fewer (let us call these *balanced* and *unbalanced* cuts respectively). If the mincut is a balanced cut, we use two random samples of $\tilde{O}(\sqrt{n})$ and $\tilde{O}(1)$ vertices each, and find (s, t)-mincuts for all pairs of vertices from the two samples. It is easy to see that w.h.p., the two samples would respectively *hit* the smaller and larger sides of the mincut, and hence, one of these (s, t)-mincuts will reveal the overall mincut of the graph.
- The main task, then, is to find the mincut when it is unbalanced. In this case, we use a sequence of steps. The first step is to use *cut sparsification* of the graph by random sampling of edges. This scales down the size of the mincut, but unlike in an undirected graph, all the cuts of a digraph do not necessarily converge to their expected values in the sample. However, crucially, *the mincut can be scaled to* $\tilde{O}(\sqrt{n})$ *while ensuring that all the unbalanced cuts converge to their expected values.*
- Since only the unbalanced cuts converge to their expected values, it is possible that some balanced cut is the new mincut of the sampled graph, having been scaled down disproportionately by the random sampling. Our next step is to *overlay* this sampled graph with an *expander* graph in the same manner as in Section 6.4. Note that an expander has a larger weight for balanced cuts than for unbalanced cuts. We choose the expansion of the graph carefully so that the balanced cuts get sufficiently large weight of edges that they are no longer candidates for the mincut of the sample, while the unbalanced cuts are only distorted by a small multiplicative factor.

- At this point, we have obtained a graph where the original mincut (which was unbalanced) is a near-mincut of the new graph. Next, we create a (fractional) packing of edge-disjoint arborescences[1] in this graph using a multiplicative weights update procedure (e.g., [105]). By duality, these arborescenes have the following property: if we sample $O(\log n)$ random arborescences from this packing, then there will be at least one arborescence w.h.p. such that the original mincut 1-respects the arborescence. (A cut 1-respects an arborescence if the latter contains just one edge from the cut.)
- Thus, our task reduces to the following: given an arborescence, find the minimum weight cut in the original graph among all those that 1-respect the arborescence. Our final technical contribution is to give an algorithm that solves this problem using $O(\log n)$ maxflow computations. For this purpose, we use a centroid-based recursive decomposition of the arborescence, where in each step, we use a set of maxflow calls that can be amortized on the original graph. The minimum cut returned by all these maxflow calls is eventually returned as the mincut of the graph.

We note that unlike both the Hao-Orlin algorithm and Gabow's algorithm that are both deterministic algorithms, our algorithm is randomized (Monte Carlo) and might yield the wrong answer with a small (inverse polynomial) probability. Derandomizing our algorithm, or matching our running time bound using a different deterministic algorithm, remains an interesting open problem.

5.2.1 Additional Preliminaries

The directed mincut problem is formally defined as follows.

Definition 5.2.1: Directed global mincut

Given a directed graph, the *global mincut* is the smallest-weight set of edges whose removal causes the graph to no longer be strongly connected.

For simplicity of notation, we define $\overline{U} = V \setminus U$ throughout this chapter. Let $\vec{\partial} U$ denote the set of edges in the cut $(U, \overline{U})$, so that the directed mincut equals $\arg\min_{\emptyset \subsetneq U \subsetneq V} w(\partial U)$.

Let $MF(m, n)$ denote the time complexity of s-t maximum flow on a digraph with n vertices and m edges. The current record for this bound is $MF(m, n) = \tilde{O}(m + n^{3/2})$ [103]. We emphasize that our directed mincut algorithm uses maxflow subroutines in a black box manner and therefore, any maxflow algorithm suffices. Correspondingly, we express our running times in terms of $MF(m, n)$.

[1]An *arborescence* is a spanning tree in a directed graph where all the edges are directed away from the root.

5.3 The Directed Mincut Algorithm

The main result of this chapter is the following:

Theorem 5.3.1: Directed mincut in $\tilde{O}(\sqrt{n})$ max-flows

There is a randomized Monte Carlo algorithm that finds a directed mincut w.h.p. in $\tilde{O}(m\sqrt{n})$ time plus $\tilde{O}(\sqrt{n})$ calls to max-flow on an n-vertex, m-edge directed graph.

We now describe the algorithm. Let S^* be the source side of a minimum cut. The algorithm considers the following two cases, computes a cut for each case and takes the smaller of the two cuts as its final output.

1. The first case aims to compute the correct mincut in the event that $\min\{|S^*|, |\overline{S^*}|\} > \theta \cdot \sqrt{n}/\log n$. In this case, we randomly sample two vertices $s, t \in V$, then with reasonable probability, they will lie on opposite sides of the mincut. In that case, we can simply compute the maxflow from s to t. Repeating the sampling $O(\sqrt{n}\log^2 n)$ times, we obtain the mincut w.h.p. The total running time for this case is $O(MF(m,n)\sqrt{n}\log^2 n)$ and is formalized in Lemma 5.3.2 below:

   Lemma 5.3.2

   If $\min\{|S^*|, |\overline{S^*}|\} > r$, then w.h.p. a mincut can be calculated in time $O(MF(m,n) \cdot (n/r) \cdot \log n)$.

   Proof. Uniformly sample a list of $k = d \cdot (n/r) \cdot \lg n$ vertices $u_1, \ldots, u_k$, where d is a large constant. Without loss of generality, assume $|S^*| \leq |\overline{S^*}|$, and let $\eta = \frac{|S^*|}{n} > \frac{r}{n}$. With probability at least $1 - 2(1-\eta)^k \geq 1 - 2e^{-k\eta} \geq 1 - 2n^{-d}$, the list $u_1, \ldots, u_k$ contains at least one vertex from each of S^* and $\overline{S^*}$. Hence, there exists i such that u_i and u_{i+1} are on different sides of the $(S^*, \overline{S^*})$ cut. By calculating maxflows for all $(u_i,\ u_{i+1})$ and $(u_{i+1},\ u_i)$ pairs, and reporting the smallest (s,t)-mincut in these calls, we return a global mincut w.h.p. □

2. The second case takes care of the event that $\min\{|S^*|, |\overline{S^*}|\} \leq \theta \cdot \sqrt{n}/\log n$. In this case, we select an arbitrary vertex s, and give an algorithm for finding an s-mincut defined as:

   Definition 5.3.3: s-mincut

   An *s-mincut* is a minimum weight cut among all those that have s on the source side of the cut, i.e., $\arg\min_{\{s\}\subseteq S \subset V} w(\partial S)$.

   Repeating this process with all edge directions reversed, and returning the smaller of the s-mincuts in the original and the reversed graphs, yields the overall mincut.

   We now describe the s-mincut algorithm, where we overload notation to denote the value of the s-mincut by λ. Here, we first guess $O(\log n)$ potential values of $\tilde{\lambda}$, which is our estimate of λ, as the powers of 2, one of which lies in the range $[\lambda, 2\lambda]$, and then for each $\tilde{\lambda}$, sparsifies the graph using Lemma 5.4.1 from Section 5.4. For each such sparsifier H, the algorithm then applies Lemma 5.5.1 from Section 5.5 to pack $O(\log n)$ s-arborescences in H in $O(m\sqrt{n}\log n)$ time, one of which will 1-respect the s-mincut in G (for the correct value of $\tilde{\lambda}$):

Definition 5.3.4: s-arborescence

An *s-arborescence* is a directed spanning tree rooted at s such that all edges are directed away from s. A directed s-cut *k-respects* an s-arborescence if there are at most k cut edges in the arborescence.

Finally, for each of the $O(\log n)$ s-arborescences, the algorithm computes the minimum s-cut that 1-respects each arborescence; this algorithm is described in Algorithm 8 and proved in Theorem 5.6.1 from Section 5.6. It runs in $O((MF(m,n)+m)\cdot\log n)$ time for each of the $O(\log n)$ arborescences.

Combining both cases, the total running time becomes $\tilde{O}(m\sqrt{n}+MF(m,n)\sqrt{n})$, which establishes Theorem 5.3.1.

5.4 Sparsification

This section aims to reduce mincut value to $\tilde{O}(\sqrt{n})$ while keeping S^* a $(1+\epsilon)$-approximate mincut for a constant $\epsilon > 0$ that we will fix later. Our algorithm in this stage has two steps. First, we use random sampling to scale down the expected value of all cuts such that the expected value of the mincut $w(\partial S^*)$ becomes $\tilde{O}(\sqrt{n})$. We also claim that ∂S^* remains an approximate mincut *among all unbalanced cuts* by using standard concentration inequalities. However, since the number of balanced cuts far exceeds that of unbalanced cuts, it might be the case that some balanced cut has now become much smaller in weight than all the unbalanced cuts. This would violate the requirement that ∂S^* should be an approximate mincut in this new graph. This is where we need our second step, where we overlay an expander on the sampled graph to raise the values of all balanced cuts above the expected value of ∂S^* while only increasing the value of ∂S^* by a small factor. This last technique is inspired by recent work of Li [69] for a deterministic mincut algorithm in undirected graphs.

Now, we prove the main property of this section:

Lemma 5.4.1

Given a digraph G, a parameter $\tilde{\lambda} \in [\lambda, 2\lambda]$, and a constant $\epsilon \in (0,1)$, we can construct in $O(m \log n)$ time a value $p \in (0,1]$ and a digraph H with $O(m)$ edges such that the following holds w.h.p. for the value $p = \min\{\frac{\sqrt{n}}{\tilde{\lambda}}, 1\}$.

1. There is a constant $\theta > 0$ (depending on ϵ) such that for any set $\emptyset \neq S \subsetneq V$ with $\min\{|S|, |\bar{S}|\} \leq \theta \cdot \sqrt{n}/\log n$, we have
$$(1-\epsilon) \cdot p \cdot \delta_G(S) \leq \delta_H(S) \leq (1+\epsilon) \cdot p \cdot \delta_G(S);$$
2. For any set $\emptyset \neq S \subsetneq V$, $\delta_H(S) \geq (1-\epsilon)p\lambda$.

Proof. If $\tilde{\lambda} \leq 2\sqrt{n}$, then $\lambda \leq \tilde{\lambda} \leq 2\sqrt{n}$ as well, so we set H to be G itself, which satisfies all the properties for $p = 1$. For the rest of the proof, we assume that $\tilde{\lambda} > 2\sqrt{n}$, so that $\lambda \geq \sqrt{n}$, and we set $p = \frac{\sqrt{n}}{\tilde{\lambda}} \leq 1$. Throughout the proof, define $\epsilon' = \epsilon/2$, $r = \frac{\epsilon'^2}{6}\sqrt{n}/\log n$, $\alpha = \frac{\sqrt{n}}{\alpha_0 r}$, and $\theta = \frac{\epsilon'^3 \alpha_0}{54}$, where α_0 is the constant from Lemma 8.2.1.

We first construct digraph $\hat{G}$ by reweighting the edges of G as follows. For each edge e in G, assign it a random new weight $w_{\hat{G}}(e)$ chosen according to binomial distribution $B(w(e), p)$. (If $w_{\hat{G}}(e) = 0$, then remove e from $\hat{G}$.) For each set $\emptyset \neq S \subsetneq V$ with $\min\{|S|, |\bar{S}|\} \leq r$, we have $\mathbb{E}\delta_{\hat{G}}(S) = p\delta_G(S)$, and by Chernoff bound, the probability that $\delta_{\hat{G}}(S)$ falls outside $[(1-\epsilon')p\delta_G(S), (1+\epsilon')p\delta_G(S)]$ is upper-bounded by $2e^{-\lambda\epsilon'^2/3} \leq 2n^{-2r}$. There are $O(n^r)$ sets S with $\min\{|S|, |\bar{S}|\} \leq r$, so by a union bound, w.h.p. all such sets satisfy $(1-\epsilon')p\delta_G(S) \leq \delta_{\hat{G}}(S) \leq (1+\epsilon')p\delta_G(S)$.

Construct graph X according to Lemma 8.2.1 and split each undirected edge into two directed edges. Let H be the "union" of $\hat{G}$ and αX, so that each edge e in H has weight $w_H(e) = w_{\hat{G}}(e) + \alpha w_X(e)$, where we say $w(e) = 0$ if e does not exist in the corresponding graph.

We now show that H satisfies the two desired properties.

1. For any set $\emptyset \neq S \subsetneq V$ with $\min\{|S|, |\bar{S}|\} \leq \theta \cdot \sqrt{n}/\log n = \frac{\epsilon'\alpha_0}{9} r \leq r$, we have $\delta_H(S) \geq \delta_{\hat{G}}(S) \geq (1-\epsilon')p\delta_G(S)$ from before, so $\delta_H(S) \geq (1-\epsilon)p\delta_G(S)$ as well. For the upper bound, we have
$$\delta_H(S) = \delta_{\hat{G}}(S) + \alpha\delta_X(S) \leq (1+\epsilon')p\delta_G(S) + 9\alpha|S| \leq (1+\epsilon')p\delta_G(S) + \epsilon'\sqrt{n} \leq (1+\epsilon)p\delta_G(S)$$
2. For any set $\emptyset \neq S \subsetneq V$ with $\min\{|S|, |\bar{S}|\} \leq \theta \cdot \sqrt{n}/\log n = \frac{\epsilon'\alpha_0}{9} r \leq r$, we have $\delta_H(S) \geq \delta_{\hat{G}}(S) \geq (1-\epsilon)p\delta_G(S) \geq (1-\epsilon)p\lambda$ as required by property (2). When $\min\{|S|, |\bar{S}|\} > r$, we have $\delta_H(S) \geq \alpha\delta_X(S) \geq \alpha\alpha_0 r \geq \sqrt{n}$ for all $\emptyset \neq S \subsetneq V$.

Finally, H has $O(m)$ edges because $E(\hat{G})$ is a subset of $E(G)$ and $E(X) = O(n)$. $\square$

5.5 Finding a 1-respecting Arborescence

In this section, we assume that there is an unbalanced mincut and show how to obtain an s-arborescence that 1-respects the mincut. More formally, we prove the following:

Lemma 5.5.1

Given weighted digraph G and a fixed vertex s such that s is in the source side of a minimum cut S^* and $\min\{|S^*|, |\overline{S^*}|\} \leq \theta \cdot \sqrt{n}/\log n$ where θ is defined in Lemma 5.4.1, in $O(m\sqrt{n}\log n)$ time we can find $O(\log n)$ s-arborescences, such that w.h.p. a minimum cut 1-respects one of them.

The idea of this lemma is as follows. First, we apply Lemma 5.4.1 to our graph G and obtain graph H. w.h.p., a mincut S^* in G corresponds to a cut in H of size $(1 \pm \epsilon)p\lambda$ and no cut in H has size less than $(1-\epsilon)p\lambda$. That is, S^* is a $(1+O(\epsilon))$-approximate mincut in H. It remains to find an arborescence in H that 1-respects S^*. To do this, we employ a multiplicative weight update (MWU) framework. The algorithm begins by setting all edge weights to be uniform (say, weight 1). Then, we repeat for $O(\sqrt{n}\log(n)/\epsilon^2)$ rounds. For each round, we find in near-linear time a minimum weight arborescence and multiplicatively increase the weight of every edge in the arborescence.

Using the fact that there is no duality gap between arborescence packing and mincut [33, 40], a standard MWU analysis implies that these arborescences that we found, after some scaling, form a $(1+\epsilon)$-approximately optimal fractional arborescence packing. So our arborescence crosses S^* at most $(1+O(\epsilon)) < 2$ times on average. Thus, if we sample $O(\log n)$ arborescences from our collections, w.h.p., one of them will 1-respect S^*.[2] Below, we formalize this high level description.

Definition 5.5.2: Packing problem [105]

For convex set $P \subseteq \mathbb{R}^n$ and nonnegative linear function $f : P \to \mathbb{R}^m$, let $\gamma^* = \min_{x\in P}\max_{j\in[m]} f_j(x)$ be the solution in P that minimizes the maximum value of $f_j(x)$ over all j, and define the *width* of the packing problem as $\omega = \max_{j\in[m],x\in P} f_j(x) - \min_{j\in[m],x\in P} f_j(x)$.

The fractional arborescence packing problem conforms to this definition. Enumerate all the s-arborescences as $A_1, A_2, \ldots, A_N$. We represent a fractional packing of arborescences as a vector in $\mathbb{R}^N$, where coordinate i represents the fractional contribution of A_i in the packing. Let$P = \{x \in \mathbb{R}^N : x^T1 = 1, x \geq 0\}$ be the convex hull of all single arborescences. For each edge j with capacity $w(j)$, $f_j(x) = \sum_{i\in[N]} x_i 1[j \in T_i]/w(j)$ is the relative load of arborescence packing x on edge j. It is easy to see that $\omega \leq 1/w_{\min}$ for tree packing. The objective function is to minimize the maximum load: $\gamma^* = \min_{x\in P}\max_{j\in[m]} f_j(x)$.

[2]This should be compared with Karger's mincut algorithm in the undirected case, where there is a factor 2 gap, and hence Karger can only guarantee a 2-respecting tree in the undirected case.

For any fractional arborescence packing $x \in \mathbb{R}^N$ with value $x^T 1 = v$ where $f_j(x) \leq 1$ for all edges j, we have $\frac{1}{v}x \in P$. In particular, the maximum arborescence packing, once scaled down by its value, is exactly the vector in P that minimizes the maximum load. Therefore, it suffices to look for the vector $x \in P$ achieving the optimal value γ^*, and then scale the vector up by $1/\gamma^*$ to obtain the maximum arborescence packing.

Next we describe the packing algorithm (Figure 2 of [105]). Maintain a vector $y \in \mathbb{R}^m$, initially set to $y = 1$. In each iteration, find $x = \arg\min_{x \in P} \sum_j y_j f_j(x)$, and then add x to set S and replace y by the vector y' defined by $y'_j = y_j(1 + \epsilon f_j(x)/\omega)$. After a number of iterations, return $\bar{x} \in P$, the average of all the vectors x over the course of the algorithm. The lemma below upper bounds the number of iterations that suffice:

Lemma 5.5.3: Corollary 6.3 of [105]

After $\lceil \frac{(1+\epsilon)\omega \ln m}{\gamma^*((1+\epsilon)\ln(1+\epsilon)-\epsilon)} \rceil$ iterations of the packing algorithm, $\bar{\gamma} = \max_j f_j(\bar{x}) \leq (1+\epsilon)\gamma^*$.

We will also make use of the (exact) duality between s-arborescence packing and minimum s-cut:

Lemma 5.5.4: Corollary 2.1 of [40]

The value of maximum s-arborescence packing is equal to the value of minimum s-cut.

Proof of Lemma 5.5.1. First, construct H according to Theorem 5.4.1. By the duality above, the minimum s-cut on H has value $\lambda_H = \frac{1}{\gamma^*}$. Since $\min\{|S^*|, |\overline{S^*}|\} \leq \theta\sqrt{n}/\log n$, we have $\lambda_H \leq \delta_H(S^*) \leq (1+\epsilon)p\lambda \leq (1+\epsilon)\sqrt{n}$.

Run the aforementioned arborescence packing algorithm up to $O(\lambda_H \ln m)$ iterations, after which Lemma 5.5.3 guarantees that $\bar{\gamma} \leq (1+\epsilon)\gamma^*$. Then $\bar{x}/\bar{\gamma}$ is a vector in P with value $1/\bar{\gamma} \geq \frac{1}{1+\epsilon}\lambda_H$.

Consider sampling a random arborescence A from the distribution specified by $\bar{x}/\bar{\gamma}$, so we choose arborescence A_i with probability $\bar{x}_i/\bar{\gamma}$. Since $\delta_H(S^*) \leq (1+\epsilon)p\lambda \leq (1+\epsilon)^2\lambda_H$, the expected number of edges in $A \cap \delta_H(S^*)$ is at most $\frac{(1+\epsilon)^2}{1-\epsilon} \leq 1+4\epsilon$ for small enough $\epsilon > 0$. Since we always have $|A \cap \delta_H(S^*)| \geq 1$, by Markov's inequality $\Pr[|A \cap \delta_H(S^*)| - 1 \geq 1] \leq 4\epsilon \leq 1/2$ for small enough ϵ. Therefore, if we uniformly sampling $O(\log n)$ arborescences from the distribution $\bar{x}/\bar{\gamma}$, at least one of the arborescences is 1-respecting w.h.p.

It remains to compute $x = \arg\min_{x \in P} \sum_j y_j f_j(x)$ on each iteration. Since $\sum_j y_j f_j(x)$ is linear in x, the minimum must be achieved by a single arborescence. So the task reduces to computing the minimum cost spanning s-arborescence, which can be done in $O(m + n\log n)$ [39]. The total time complexity, over all iterations, becomes $O((m + n\log n)\lambda_H \log n) = O((m + n\log n)\sqrt{n}\log n)$. $\square$

5.6 Mincut Given 1-respecting Arborescence

We propose an algorithm (Algorithm 8) that uses $O(\log n)$ maxflow subroutines to find the minimum s-cut that 1-respects a given s-arborescence. The result is formally stated in Theorem 5.6.1.

Theorem 5.6.1

Consider a directed graph $G = (V, E, w)$ with n vertices, m edges, and polynomially bounded edge weights $w_e > 0$. Fix a global (directed) mincut S of G. Given an arborescence T rooted at $s \in S$ with $|T \cap (S, \overline{S})| = 1$, Algorithm 8 outputs a global minimum cut of G in time $O((MF(m, n) + m) \cdot \log n)$.

We first give some intuition for Algorithm 8. Because $s \in S$, if we could find a vertex $t \in \overline{S}$, then computing the s-t mincut using one maxflow call would yield a global mincut of G. However, we cannot afford to run one maxflow between s and every other vertex in G. Instead, we carefully partition the vertices into $\ell = O(\log n)$ sets $(C_i)_{i=1}^{\ell}$. We show that for each C_i, we can modify the graph appropriately so that it allows us to (roughly speaking) compute the maximum flow between s and every vertex $c \in C_i$ using one maxflow call.

More specifically, Algorithm 8 has two stages. In the first stage, we compute a *centroid decomposition* of T. Recall that a centroid of T is a vertex whose removal disconnects T into subtrees with at most $n/2$ vertices. This process is done recursively, starting with the root s of T. We let P_1 denote the subtrees resulting from the removal of s from T. In each subsequent step i, we compute the set C_i of the centroids of the subtrees in P_i. We then remove the centroids and add the resulting subtrees to P_{i+1}. This process continues until no vertices remain.

In the second stage, for each layer i, we construct a directed graph G_i and perform one maxflow computation on G_i. The maxflow computation on G_i would yield candidate cuts for every vertex in C_i, and after computing the appropriate maximum flow across every layer, we output the minimum candidate cut as the minimum cut of G. The details are presented in Algorithm 8.

We first state two technical lemmas that we will use to prove Theorem 5.6.1.

Lemma 5.6.2

Recall that P_i is the set of subtrees in layer i and C_i contains the centroid of each subtree in P_i. If $C_j \subseteq S$ for every $0 \le j < i$, then $\overline{S}$ is contained in exactly one subtree in P_i, and consequently, at most one vertex $u \in C_i$ can be in $\overline{S}$.

Algorithm 8 Finding the global minimum directed cut.

```
Input: An arborescence $T$ rooted at $s \in S$ such that $S$ 1-respects $T$.
 1: // Stage I: Build centroid decomposition.
 2: Let $C_0 = \{s\}$, $P_1 =$ the set of subtrees obtained by removing $s$ from $T$, and $i = 1$.
 3: while $P_i \neq \varnothing$ do
 4:     Initialize $C_i$ (the centroids of $P_i$) and $P_{i+1}$ as empty sets.
 5:     for each subtree $U \in P_i$ do Compute the centroid $u$ of $U$ and add it to $C_i$.
 6:         Add all subtrees generated by removing $u$ from $U$ to $P_{i+1}$.
 7:     Set $\ell = i$ and iterate $i = i + 1$.
 8: // Stage II: Calculate integrated maximum flow for each layer.
 9: for $i = 1$ to $\ell$ do
10:     Construct a digraph $G_i = (V \cup \{t_i\}, E_1 \cup E_2 \cup E_3)$ as follows (see Figure 5.1):
11:       1) Add edges $E_1 = E \cap \cup_{U \in P_i}(U \times U)$ with capacity equal to their original weight.
12:       2) Add edges $E_2 = \{(s, v) : (u, v) \in E \setminus E_1\}$ with capacity of $(s, v)$ equal to the
    original weight of $(u, v)$.
13:       3) Add edges $E_3 = \{(u, t_i) : u \in C_i\}$ with infinite capacity.
14:     Compute the maximum $s$-$t_i$ flow $f_i^*$ in $G_i$.
15:     For each component $U \in P_i$ with centroid $u$, the value of $f_i^*$ on edge $(u, t_i)$ is a
    candidate cut value, and the nodes in $U$ that can reach $u$ in the residue graph is a
    candidate for $\overline{S}$.
16: Return the smallest candidate cut as minimum $s$-cut and the corresponding $(S, \overline{S})$.
```

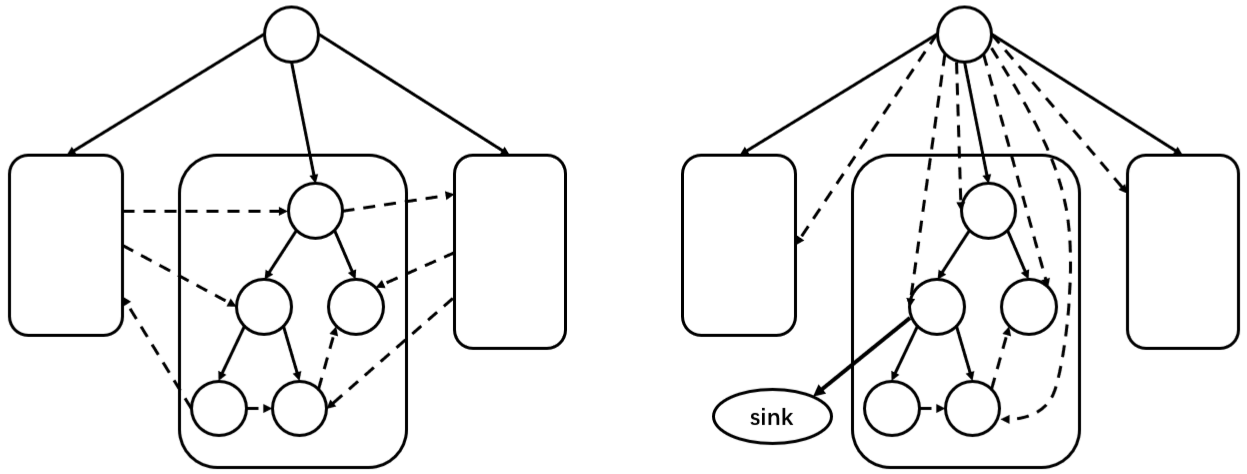

Figure 5.1: Construction of auxiliary graph G_i in Algorithm 8. Solid lines represent the arborescence T. Dashed lines are other edges in the graph. Rectangles are sets formed by the first level of centroid decomposition. Left: The original graph. Right: The part of G_1 solving the case that the mincut separates root and the centroid of the middle subtree.

Lemma 5.6.3

Let G_i be the graph constructed in Step 10 of Algorithm 8. Let f_i^* be a maximum s-t_i flow on G_i as in Step 14. For any $U \in P_i$ with centroid u, the amount of flow f_i^* puts on edge (u, t_i) is equal to the value of the minimum cut between $\overline{U}$ and u.

We defer the proofs of Lemmas 5.6.2 and 5.6.3, and first use them to prove Theorem 5.6.1.
Proof of Theorem 5.6.1. We first prove the correctness of Algorithm 8.

Because $C_0 = \{s\}$ and $s \in S$, and the C_i's form a disjoint partition of V, there must be a layer i such that for the first time, we have a centroid $u \in C_i$ that belongs to $\overline{S}$. By Lemma 5.6.2, we know that $\overline{S}$ must be contained in exactly one subtree $U \in P_i$, and hence u must be the centroid of U. In summary, we have $u \in \overline{S}$ and $\overline{S} \subseteq U$.

Consider the graph G_i constructed for layer i. By Lemma 5.6.3, based on the flow f_i^* puts on the edge (u, t_i), we can recover the value of the minimum (directed) cut between $\overline{U}$ and u. Because $\overline{S} \subseteq U$ (or equivalently $\overline{U} \subseteq S$) and $u \in \overline{S}$, the cut $(S, \overline{S})$ is one possible cut that separates $\overline{U}$ and u. Therefore, the flow that f_i^* puts on the edge (u, t_i) is equal to the global mincut value in G.

In addition, the candidate cut value for any other centroid u' of a subtree $U' \in P_i$ must be at least the mincut value between s and u'. This is because the additional restriction that the cut has to separate $\overline{U'}$ from u' can only make the mincut value larger, and the value of this cut in G_i is equal to the value of the same cut in G. Therefore, the minimum candidate cut value in all ℓ layers must be equal to the global mincut value of G.

Now we analyze the running time of Algorithm 8. We can find the centroid of an n-node tree in time $O(n)$ (see e.g., [81]). The total number of layers $\ell = O(\log n)$ because removing the centroids reduces the size of the subtrees by at least a factor of 2. Thus, the running time of Stage I of Algorithm 8 is $O(n \log n)$. In Stage II, we can construct each G_i in $O(m)$ time and every G_i has $O(m)$ edges. Since there are $O(\log n)$ layers and the maximum flow computations take a total of $O(MF(m, n) \cdot \log n)$ time, the overall runtime is $O(n \log n + (MF(m, n) + m) \log n) = O((MF(m, n) + m) \log n)$. $\square$

Before proving Lemmas 5.6.2 and 5.6.3 we first prove the following lemma.

Lemma 5.6.4

If x and y are vertices in $\overline{S}$, then every vertex on the (undirected) path from x to y in the arborescence T also belongs to $\overline{S}$.

Proof. Consider the lowest common ancestor z of x and y. Because there is a directed path from z to x and a directed path from z to y, we must have $z \in \overline{S}$. Otherwise, there are at least two edges in T that go from S to $\overline{S}$.

Because $s \in S$ and $z \in \overline{S}$, there is already an edge in T (on the path from s to z) that goes from S to $\overline{S}$. Consequently, all other edges in T cannot go from S to $\overline{S}$, which means the entire path from z to x (and similarly z to y) must be in $\overline{S}$. $\square$

Recall that Lemma 5.6.2 states that if all the centroids in previous layers are in S, then $\overline{S}$ is contained in exactly one subtree U in the current layer i.

Proof of Lemma 5.6.2. For contradiction, suppose that there exist distinct subtrees U_1 and U_2 in P_i and vertices $x, y \in \overline{S}$ such that $x \in U_1$ and $y \in U_2$.

By Lemma 5.6.4, any vertex on the (undirected) path from x to y also belongs to $\overline{S}$. Consider the first time that x and y are separated into different subtrees. This must have happened because some vertex on the path from x to y is removed. However, the set of vertices removed at this point of the algorithm is precisely $\bigcup_{0\leq j<i} C_j$, but our hypothesis assumes that none of them are in $\overline{S}$. This leads to a contradiction and therefore $\overline{S}$ is contained in exactly one subtree of P_i.

It follows immediately that at most one centroid $u \in C_i$ can be in $\overline{S}$. □

Next we prove Lemma 5.6.3, which states that the maximum flow between s and t_i in the modified graph G_i allows one to simultaneously compute a candidate mincut value for each vertex $u \in C_i$.

Proof of Lemma 5.6.3. First observe that the maxflow computation from s to t_i in G_i can be viewed as multiple independent maxflow computations. The reason is that, for any two subtrees $U_1, U_2 \in P_i$, there are only edges that go from s into U_1 and from U_1 to t_i in G_i (similarly for U_2), but there are no edges that go between U_1 and U_2.

The above observation allows us to focus on one subtree $U \in P_i$. Consider the procedure that we produce G_i from G in Steps 11 to 13 of Algorithm 8. The edges with both ends in U are intact (the edge set E_1). If we contract all vertices out of U into s, then all edges that enter U would start from s, which is precisely the effect of removing cross-subtree edges and adding the edges in E_2. One final infinity-capacity edge $(u, t_i) \in E_3$ connects the centroid of U to the super sink t_i.

Therefore, the maximum s-t_i flow f_i^* computes the maximum flow between $\overline{U}$ and $u \in U$ simultaneously for all $U \in P_i$, whose value is reflected on the edge (u, t_i). It follows from the maxflow mincut theorem that the flow on edge (u, t_i) is equal to the mincut value between $\overline{U}$ and u in G (i.e., the minimum value $w(A, \overline{A})$ among all $A \subset V$ with $\overline{U} \subseteq A$ and $u \in \overline{A}$). □

5.7 Conclusion

In this section, we presented our directed mincut algorithm which runs in roughly $\tilde{O}(\sqrt{n})$ many max-flows. There is still a lot of progress to be made, and we expect polylogarithmic many max-flows to be eventually within reach. Improving upon our result would likely require a more sophisticated sparsification procedure which may involve dependent sampling among edges.

Another interesting direction to take is trying to avoid max-flow computations altogether, which is possible for undirected global mincut. This would likely require a fundamentally different approach, however, as even the sparsification and arborescence packing part of our algorithm—which were directly inspired by Karger's in the undirected case—could not avoid max-flow computations.

Acknowledgements

The source of Theorem 5.3.1 [18] is a merge of two independent results that are unpublished on their own. Section 5.4 is due to Jason Li, Danupon Nanongkai, and Thatchaphol Saranurak, and Sections 5.5 and 5.6 is by Ruoxu Cen and Debmalya Panigrahi.

Part II

Preconditioning

Chapter 6

Deterministic Mincut

This chapter is dedicated to the global mincut problem: given an undirected graph, find the minimum-weight set of edges whose removal disconnects the graph. We study global mincut from a preconditioning point of view and present the first almost-linear deterministic algorithm for this problem, following the work of [69].

Prior to the works studied in this thesis, the global mincut problem was one of the most notorious to derandomize in the context of graph algorithms. Even though a near-linear time randomized algorithm was known since the 1990s [55], the best deterministic running time for general graphs remained $\tilde{O}(mn)$. Karger's algorithm was difficult to derandomize due to its *sparsification* step, where we sparsify the graph to approximately preserve the global mincut.

In this chapter, we manage to derandomize Karger's graph sparsification routine through a preconditioning approach: we first restrict our attention to the expander graphs, and then apply expander decompositions to handle the general case. To convey the conceptual insights of this chapter in a relatively simple setting, we devote a separate section, Section 6.4, for just the expander case. There, we discuss a key property of expanders that we exploit in our algorithm: the global mincut in an expander must be *unbalanced*; in other words, we can employ a locality-based algorithm to solve this case. This shows that the concepts of preconditioning and locality often go hand-in-hand. Finally, we handle the general case by computing a recursive expander decomposition of the graph, which is covered in Section 6.5 and constitutes the technical bulk of the chapter.

6.1 Background

The global mincut problem dates back to the work of Gomory and Hu [44] in 1961 who gave an algorithm to compute the mincut of an n-vertex graph using $n-1$ max-flow computations. Since then, a large body of research has been devoted to obtaining faster algorithms for this problem. In 1992, Hao and Orlin [48] gave a clever amortization of the $n-1$ max-flow computations to match the running time of a single max-flow computation, though their method is specific to the push-relabel max-flow algorithm [42] and therefore takes $O(mn\log(n^2/m))$ time. Around the same time, Nagamochi and Ibaraki [86] (see also [85]) designed an algorithm that bypasses max-flow computations altogether, a technique that was further refined by Stoer and Wagner [102] (and independently by Frank in unpublished work). This alternative method yields a running time of $O(mn+n^2\log n)$. Before the results of this thesis, these $\tilde{O}(mn)$ time algorithms were the fastest *deterministic* mincut algorithms for weighted graphs.

Starting with Karger's contraction algorithm in 1993 [54], a parallel body of work started to emerge in *randomized* algorithms for the mincut problem. This line of work (see also Karger and Stein [56]) eventually culminated in a breakthrough paper by Karger [55] in 1996 that gave an $O(m\log^3 n)$ time *Monte Carlo* algorithm for the mincut problem. Note that this algorithm comes to within poly-logarithmic factors of the optimal $O(m)$ running time for this problem. In that paper, Karger asks whether we can also achieve near-linear running time using a *deterministic* algorithm, which we answer affirmatively in this chapter.

Karger's question has also been resolved for specific instances in the past. In a recent breakthrough, Kawarabayashi and Thorup [57] gave the first near-linear time deterministic algorithm for this problem *for simple graphs*. They obtained a running time of $O(m\log^{12} n)$, which was later improved by Henzinger, Rao, and Wang [51] to $O(m\log^2 n\log\log^2 n)$, and then simplified by Saranurak [93] at the cost of $m^{1+o(1)}$ running time. From a technical perspective, Kawarabayashi and Thorup's work introduced the idea of using low conductance cuts to find the mincut of the graph, the main inspiration behind the preconditioning-based approach we present in this chapter.

6.2 Our Techniques

Our main result is formally stated as follows.

Theorem 6.2.1: Deterministic mincut

There is a deterministic algorithm that computes the mincut of a weighted, undirected graph in $2^{O(\log n)^{5/6}(\log\log n)^{O(1)}}m$ time.

At a high level, we follow Karger's approach and essentially de-randomize the single randomized procedure in Karger's near-linear time mincut algorithm [55], namely the construction of the *skeleton* graph, which Karger accomplishes through the Benczur-Karger graph sparsification technique by random sampling. We remark that our de-randomization does not recover a full $(1+\epsilon)$-approximate graph sparsifier, but the skeleton graph that we obtain is sufficient to solve the mincut problem.

Let us first briefly review the Benczur-Karger graph sparsification technique, and discuss the difficulties one encounters when trying to de-randomize it. Given a weighted, undirected graph, the sparsification algorithm samples each edge independently with a probability depending on the weight of the edge and the global mincut of the graph, and then re-weights the sampled edge accordingly. In traditional graph sparsification, we require that every cut in the graph has its weight preserved up to a $(1+\epsilon)$ factor. There are exponentially many cuts in a graph, so a naive union bound over all cuts does not work. Benczur and Karger's main insight is to set up a more refined union bound, layering the (exponentially many) cuts in a graph by their weight. They show that for all $\alpha \ge 1$, there are only $n^{c\alpha}$ many cuts in a graph whose weight is roughly α times the mincut, and each one is preserved up to a $(1+\epsilon)$ factor with probability $1 - n^{-c'\alpha}$, for some constants $c' \gg c$. In other words, they establish a union bound layered by the α-approximate mincuts of a graph, for each $\alpha \ge 1$.

One popular method to de-randomize random sampling algorithms is through *pessimistic estimators*, which is a generalization of the well-known method of conditional probabilities. For the graph sparsification problem, the method of pessimistic estimators can be implemented as follows. The algorithm considers each edge one by one in some arbitrary order, and decides on the spot whether to keep or discard each edge for the sparsifier. To make this decision, the algorithm maintains a *pessimistic estimator*, which is a real number in the range $[0, 1)$ that represents an upper bound on the probability of failure should the remaining undecided edges each be sampled independently at random. In many cases, the pessimistic estimator is exactly the probability upper bound that one derives from analyzing the random sampling algorithm, except conditioned on the edges kept and discarded so far. The algorithm makes the choice—whether to keep or discard the current edge—based on whichever outcome does not increase the pessimistic estimator; such a choice must always exist for the pessimistic estimator to be valid. Once all edges are processed, the pessimistic estimator must still be a real number less than 1. But now, since there are no more undecided edges, the probability of failure is either 0 or 1. Since the pessimistic estimator is an upper bound which is less than 1, the probability of failure must be 0; in other words, the set of chosen edges is indeed a sparsifier of the graph.

In order for this de-randomization procedure to be efficient, the pessimistic estimator must be quickly evaluated and updated after considering each edge. Unfortunately, the probability union bound in the Benczur-Karger analysis involves all cuts in the graph, and is therefore an expression of exponential size and too expensive to serve as our pessimistic estimator. To design a more efficient pessimistic estimator, we need a more compact, easy-to-compute union bound over all cuts of the graph. We accomplish this by grouping all cuts of the graph into two types: small cuts and large cuts.

Small cuts. Recall that our goal is to preserve cuts in the graph up to a $(1+\epsilon)$ factor. Let us first restrict ourselves to all α-approximate mincuts of the graph for some $\alpha = n^{o(1)}$. There can be $n^{\Omega(\alpha)}$ many such cuts, so the naive union bound is still too slow. Here, our main strategy is to establish a *structural representation* of all α-approximate mincuts of a graph, with the goal of deriving a more compact "union bound" over all α-approximate cuts. For an expander, this task is relatively easy: in an expander with conductance ϕ, all α-approximate mincuts must have at most α/ϕ vertices on one side, so a compact representation is simply all cuts with at most α/ϕ vertices on one side. Motivated by this connection, we show that if the original graph is itself an expander, then it is enough to preserve all vertex degrees and all edge weights up to an additive $\epsilon'\lambda$ factor, where λ is the mincut of the graph and ϵ' depends on ϵ, α, ϕ. We present the unweighted expander case in Section 6.4 as a warm-up, which features all of our ideas except for the final expander decomposition step. To handle general graphs, we compute an *expander hierarchy* of the graph, which is a recursive, hierarchical expander decomposition structure introduced by Goranci et al. [45].

Large cuts. For the large cuts—those that are not α-approximate mincuts—our strategy differs from the pessimistic estimator approach. Here, our aim is not to preserve each of them up to a $(1+\epsilon)$-factor, but a γ-factor for a different parameter $\gamma = n^{o(1)}$. This relaxation prevents us from obtaining a full $(1+\epsilon)$-approximate sparsification of the graph, but it still works for the mincut problem since the large cuts do not fall below the original mincut value. While a deterministic $(1+\epsilon)$-approximate sparsification algorithm in near-linear time is unknown, one exists for γ-approximation sparsification for some $\gamma = n^{o(1)}$ [27]. In our case, we actually need the sparsifier to be *uniformly weighted*, so we construct our own sparsifier in Section 6.5.3, again via the expander hierarchy. Note that if the original graph is an expander, then we can take any expander whose degrees are roughly the same; in particular, the sparsifier does not need to be a subgraph of the original graph. To summarize, for the large cuts case, we simply construct an γ-approximate sparsifier deterministically, bypassing the need to de-randomize the Benczur-Karger random sampling technique.

Combining them together. Of course, this γ-approximate sparsifier destroys the guarantee of the small cuts, which need to be preserved $(1+\epsilon)$-approximately. Our strategy is to combine the small cut sparsifier and the large cut sparsifier together in the following way. We take the union of the small cut sparsifier with a "lightly" weighted version of the large cut sparsifier, where each edge in it is weighted by ϵ/γ times its normal weight. This way, each small cut of weight w suffers at most an additive $\gamma w \cdot \epsilon/\gamma = \epsilon w$ weight from the "light" large cut sparsifier, so we do not destroy the small cuts guarantee (up to replacing ϵ with 2ϵ). Moreover, each large cut of weight $w \geq \alpha\lambda$ is weighted by at least $w/\gamma \cdot \epsilon/\gamma \geq \alpha\lambda/\gamma \cdot \epsilon/\gamma = \alpha/\gamma^2 \cdot \epsilon\lambda$, where λ is the mincut of the original graph. Hence, as long as $\alpha \geq \gamma^2/\epsilon$, the large cuts have weight at least the mincut, and the property for large cuts is preserved.

Unbalanced vs. balanced. We remark that our actual separation between small cuts and large cuts is somewhat different; we use *unbalanced* and *balanced* instead to emphasize this distinction. Nevertheless, we should intuitively think of unbalanced cuts as having small weight and balanced as having large weight; rather, the line is not drawn precisely at a weight threshold of $\alpha\lambda$. The actual separation is more technical, so we omit it in this overview section.

6.3 Additional Preliminaries

In this chapter, sometimes we will mention weighted graphs and unweighted graphs in the same context. We make this distinction because sometimes, it is necessary to consider unweighted graphs on their own, rather than as a special case of weighted graphs. In particular, the *skeleton* graph that we compute in the algorithm *must* be unweighted. For this reason, we introduce separate graph-theoretic notation for unweighted graphs to better differentiate them from their weighted counterparts.

For an unweighted graph $G = (V, E)$, let $\#(u, v)$ be the number of (parallel) edges $e \in E$ with endpoints u and v. For a set $F \subseteq E$ of edges, denote its cardinality by $|F|$, and for a vertex $v \in V$, define its degree $\deg(v)$ to be $|\partial(\{v\})|$. The rest of the definitions remain the same for weighted and unweighted graphs.

6.3.1 Karger's Approach

In this section, we outline Karger's approach to his near-linear time randomized mincut algorithm and set up the necessary theorems for our deterministic result. Karger's algorithm has two main steps. First, it computes a small set of (unweighted) trees on vertex set V such that the mincut 2-*respects* one of the trees T, defined as follows:

Definition 6.3.1

Given a weighted graph G and an unweighted tree T on the same set of vertices, a cut $\partial_G S$ 2-*respects* the tree T if $|\partial_T S| \leq 2$.

Karger accomplishes this goal by first *sparsifying* the graph into an unweighted *skeleton* graph using the well-known Benzcur-Karger sparsification by random sampling, and then running a *tree packing* algorithm of Gabow [40] on the skeleton graph.

Theorem 6.3.2: Global mincut given skeleton graph [55]

Let G be a weighted graph, let m' and c' be parameters, and let H be an unweighted graph on the same vertices, called the *skeleton* graph, with the following properties:

(a) H has m' edges,

(b) The mincut of H is c', and

(c) The mincut in G corresponds (under the same vertex partition) to a $7/6$-approximate mincut in H.

Given graphs G and H, there is a deterministic algorithm in $O(c'm' \log n)$ time that constructs $O(c')$ trees on the same vertices such that one of them 2-respects the mincut in G.

The second main step of Karger's algorithm is to compute the mincut of G given a tree that 2-respects the mincut. This step is deterministic and is based on dynamic programming.

Theorem 6.3.3: Minimum 2-respecting cut algorithm [55]

Given a weighted, undirected graph G and a (not necessarily spanning) tree T on the same vertices, there is a deterministic algorithm in $O(m \log^2 n)$ time that computes the minimum-weight cut in G that 2-respects the tree T.

Our main technical contribution is a deterministic construction of the skeleton graph used in Theorem 6.3.2. Instead of designing an algorithm to produce the skeleton graph directly, it is more convenient to prove the following, which implies a skeleton graph by the following claim.

Theorem 6.3.4: Mincut sparsifier

For any $0 < \epsilon \leq 1$, we can compute, in deterministic $\epsilon^{-4} 2^{O(\log n)^{5/6}(\log\log n)^{O(1)}} m$ time, an unweighted graph H and some weight $W = \epsilon^4 \lambda / 2^{O(\log n)^{5/6}(\log\log n)^{O(1)}}$ such that

1. For any mincut ∂S^* of G, we have $W \cdot |\partial_H S^*| \leq (1+\epsilon)\lambda$, and
2. For any cut $\emptyset \subsetneq S \subsetneq V$ of G, we have $W \cdot |\partial_H S| \geq (1-\epsilon)\lambda$.

Claim 6.3.5

For $\epsilon = 0.01$, the graph H in Theorem 6.3.4 fulfills the conditions of Theorem 6.3.2 with $m' = m^{1+o(1)}$ and $c' = n^{o(1)}$.

Proof. Since the algorithm of Theorem 6.3.4 takes $m^{1+o(1)}$ time, the output graph H must have $m^{1+o(1)}$ edges, fulfilling condition (a) of Theorem 6.3.2. For any mincut S^* of G, by property (1) of Theorem 6.3.4, we have $|\partial_H S^*| \leq (1+\epsilon)\lambda/W \leq n^{o(1)}$, fulfilling condition (b). For any cut $\emptyset \subsetneq S \subsetneq V$, by property (2), we have $|\partial_H S| \geq (1-\epsilon)\lambda/W$. In other words, S^* is a $(1+\epsilon)/(1-\epsilon)$-approximate mincut, which is a $7/6$-approximate mincut for $\epsilon = 0.01$, fulfilling condition (c). $\square$

With the above three statements in hand, we now prove Theorem 6.2.1 following Karger's approach. Run the algorithm of Theorem 6.3.4 to produce a graph H which, by Claim 6.3.5, satisfies the conditions of Theorem 6.3.2. Apply Theorem 6.3.2 on G and the skeleton graph H, producing $n^{o(1)}$ many trees such that one of them 2-respects the mincut in G. Finally, run Theorem 6.3.3 on each tree separately and output the minimum 2-respecting cut found among all the trees, which must be the mincut in G. Each step requires $2^{O(\log n)^{5/6}(\log\log n)^{O(1)}} m$ deterministic time, proving Theorem 6.2.1.

Thus, the main focus for the rest of the chapter is proving Theorem 6.3.4.

6.3.2 Spectral Graph Theory

Central to our approach are the well-known concepts of *conductance*, *expanders*, and the graph *Laplacian* from spectral graph theory.

Definition 6.3.6: Conductance, expander

The *conductance* of a weighted graph G is

$$\Phi(G) := \min_{\emptyset \subsetneq S \subsetneq V} \frac{w(E(S, V \setminus S))}{\min\{\mathbf{vol}(S), \mathbf{vol}(V \setminus S)\}}.$$

For the conductance of an unweighted graph, replace $w(E(S, V \setminus S))$ by $|E(S, V \setminus S)|$. We say that G is a *ϕ-expander* if $\Phi(G) \geq \phi$.

Definition 6.3.7: Laplacian

The *Laplacian* L_G of a weighted graph $G = (V, E)$ is the $n \times n$ matrix, indexed by $V \times V$, where

(a) Each diagonal entry (v, v) has entry $\deg(v)$, and

(b) Each off-diagonal entry (u, v) $(u \neq v)$ has weight $-w(u, v)$ if $(u, v) \in E$ and 0 otherwise.

The only fact we will use about Laplacians is the following well-known fact, that cuts in graphs have the following nice form:

Fact 6.3.8

For any weighted graph $G = (V, E)$ with Laplacian L_G, and for any subset $S \subseteq V$, we have

$$w(\partial S) = \mathbb{1}_S^T L_G \mathbb{1}_S,$$

where $\mathbb{1}_S \in \{0, 1\}^V$ is the vector with value 1 at vertex v if $v \in S$, and value 0 otherwise. For unweighted graph G, replace $w(\partial S)$ with $|\partial S|$.

6.4 Expander Case

In this section, we prove Theorem 6.3.4 restricted to the case when G is an *unweighted expander*. Our aim is to present an informal, intuitive exposition that highlights our main ideas in a relatively simple setting. Since this section is not technically required for the main result, we do not attempt to formalize our arguments, deferring the rigorous proofs to the general case in Section 6.5.

Theorem 6.4.1: Deterministic mincut on expanders

Let G be an unweighted ϕ-expander multigraph. For any $0 < \epsilon \le 1$, we can compute, in deterministic $m^{1+o(1)}$ time, an unweighted graph H and some weight $W = \epsilon^3\lambda/n^{o(1)}$ such that

(a) For any mincut $\partial_G S^*$ of G, we have $W \cdot |\partial_H S^*| \le (1+\epsilon)\lambda$, and

(b) For any cut $\partial_G S$ of G, we have $W \cdot |\partial_H S| \ge (1-\epsilon)\lambda$.

For the rest of this section, we prove Theorem 6.4.1.

Consider an arbitrary cut $\partial_G S$. By Fact 6.3.8, we have

$$|\partial_G S| = \mathbb{1}_S^T L_G \mathbb{1}_S = \left(\sum_{v\in S} \mathbb{1}_v^T\right) L_G \left(\sum_{v\in S} \mathbb{1}_v\right) = \sum_{u,v\in S} \mathbb{1}_u^T L_G \mathbb{1}_v. \tag{6.1}$$

Suppose we can approximate each $\mathbb{1}_u^T L_G \mathbb{1}_v$ to an additive error of $\epsilon'\lambda$ for some small ϵ' (depending on ϵ); that is, suppose that our graph H and weight W satisfy

$$|\mathbb{1}_u^T L_G \mathbb{1}_v - W \cdot \mathbb{1}_u^T L_H \mathbb{1}_v| \le \epsilon'\lambda$$

for all $u, v \in V$. Then, by (6.1), we can approximate $|\partial_G S|$ up to an additive $|S|^2\epsilon'\lambda$, or a multiplicative $(1 + |S|^2\epsilon')$, which is good if $|S|$ is small. Similarly, if $|V \setminus S|$ is small, then we can replace S with $V \setminus S$ in (6.1) and approximate $|\partial_G S| = |\partial_G(V \setminus S)|$ to the same factor. Motivated by this observation, we define a set $S \subseteq V$ to be *unbalanced* if $\min\{\mathbf{vol}(S), \mathbf{vol}(V \setminus S)\} \le \alpha\lambda/\phi$ for some $\alpha = n^{o(1)}$ to be set later. Similarly, define a cut $\partial_G S$ to be unbalanced if the set S is unbalanced. Note that an unbalanced set S must have either $|S| \le \alpha/\phi$ or $|V \setminus S| \le \alpha/\phi$, since if we assume without loss of generality that $\mathbf{vol}(S) \le \mathbf{vol}(V \setminus S)$, then

$$|S|\lambda \le \sum_{v\in S} \deg(v) = \mathbf{vol}(S) \le \alpha\lambda/\phi, \tag{6.2}$$

where the first inequality uses that each degree cut $\partial(\{v\})$ has weight $\deg(v) \ge \lambda$. Moreover, since G is a ϕ-expander, the mincut $\partial_G S^*$ is unbalanced because, assuming without loss of generality that $\mathbf{vol}(S^*) \le \mathbf{vol}(V \setminus S^*)$, we obtain

$$\frac{|\partial_G(S^*)|}{\mathbf{vol}(S^*)} \ge \Phi(G) \ge \phi \implies \mathbf{vol}(S^*) \le 1/\phi \le \alpha\lambda/\phi.$$

To approximate all unbalanced cuts, it suffices by (6.1) and (6.2) to approximate each $\mathbb{1}_u^T L_G \mathbb{1}_v$ up to additive error $(\phi/\alpha)^2\epsilon\lambda$. When $u \ne v$, the expression $\mathbb{1}_u^T L_G \mathbb{1}_v$ is simply the negative of the number of parallel (u, v) edges in G. So, approximating $\mathbb{1}_u^T L_G \mathbb{1}_v$ up to additive error $\epsilon\lambda$ simply amounts to approximating the number of parallel (u, v) edges. When $u = v$, the expression $\mathbb{1}_v^T L_G \mathbb{1}_v$ is simply the degree of v, so approximating it amounts to approximating the degree of v.

Consider what happens if we randomly sample each edge with probability $p = \Theta(\frac{\alpha \log n}{\epsilon^2 \phi \lambda})$ and weight the sampled edges by $\widehat{W} := 1/p$ to form the sampled graph $\widehat{H}$. For the terms $\mathbb{1}_u^T L_G \mathbb{1}_v$ $(u \neq v)$, we have $\#_G(u,v) \le \mathbf{vol}(S) \le \alpha\lambda/\phi$. Let us assume for simplicity that $\#_G(u,v) = \alpha\lambda/\phi$, which turns out to be the worst case. By Chernoff bounds, for $\delta = \epsilon\phi/\alpha$,

$$\begin{aligned}\Pr\left[\left|\#_{\widehat{H}}(u,v) - p\cdot\#_G(u,v)\right| > \delta\cdot p\cdot\#_G(u,v)\right] &< 2\exp(-\delta^2\cdot p\cdot\#_G(u,v)/3)\\ &= 2\exp\left(-\left(\frac{\epsilon\phi}{\alpha}\right)^2\cdot\Theta\left(\frac{\alpha\log n}{\epsilon^2\phi\lambda}\right)\cdot\frac{\alpha\lambda/\phi}{3}\right) \qquad (6.3)\\ &= 2\exp(-\Theta(\log n)),\end{aligned}$$

which we can set to be much less than $1/n^2$. We then have the implication

$$\begin{aligned}\left|\#_{\widehat{H}}(u,v) - p\cdot\#_G(u,v)\right| \le \delta\cdot p\cdot\#_G(u,v) \implies \left|\mathbb{1}_u^T(L_G - L_{\widehat{H}})\mathbb{1}_v\right| &\le \delta\cdot\#_G(u,v)\\ &= \epsilon\phi/\alpha\cdot\alpha\lambda/\phi = \epsilon\lambda.\end{aligned}$$

Similarly, for the terms $\mathbb{1}_v^T L_G \mathbb{1}_v$, we have $\deg(v) \le \mathbf{vol}(S) \le \alpha\lambda/\phi$, and the same calculation can be made.

From this random sampling analysis, we can derive the following pessimistic estimator. Initially, it is the sum of the quantities (6.3) for all (u,v) satisfying either $u = v$ or $(u,v) \in E$. This sum has $O(m)$ terms which sum to less than 1, so it can be efficiently computed and satisfies the initial condition of a pessimistic estimator. After some edges have been considered, the probability upper bounds (6.3) are modified to be conditional to the choices of edges so far, which can still be efficiently computed. At the end, for each unbalanced set S, the graph $\widehat{H}$ will satisfy

$$\left||\partial_G S| - \widehat{W}\cdot|\partial_{\widehat{H}} S|\right| \le \epsilon\lambda \implies (1-\epsilon)|\partial_G S| \le \widehat{W}\cdot|\partial_{\widehat{H}} S| \le (1+\epsilon)|\partial_G S|.$$

Since any mincut $\partial_G S^*$ is unbalanced, we fulfill condition (a) of Theorem 6.4.1. We also fulfill condition (b) for any cut with a side that is unbalanced. This concludes the unbalanced case; we omit the rest of the details, deferring the pessimistic estimator and its efficient computation to the general case, specifically Section 6.5.2.

Define a cut to be *balanced* if it is not unbalanced. For the balanced cuts, it remains to fulfill condition (b), which may not hold for the graph $\widehat{H}$. Our solution is to "overlay" a fixed expander onto the graph $\widehat{H}$, weighted small enough to barely affect the mincut (in order to preserve condition (a)), but large enough to force all balanced cuts to have weight at least λ. In particular, let $\widetilde{H}$ be an unweighted $\Theta(1)$-expander on the same vertex set V where each vertex $v \in V$ has degree $\Theta(\deg_G(v)/\lambda)$, and let $\widetilde{W} := \Theta(\epsilon\phi\lambda)$. We should think of $\widetilde{H}$ as a "lossy" sparsifier of G, in that it approximates cuts up to factor $O(1/\phi)$, not $(1+\epsilon)$.

Consider taking the "union" of the graph $\widehat{H}$ weighted by $\widehat{W}$ and the graph $\widetilde{H}$ weighted by $\widetilde{W}$. More formally, consider a weighted graph H' where each edge (u,v) is weighted by $\widehat{W} \cdot w_{\widehat{H}}(u,v) + \widetilde{W} \cdot w_{\widetilde{H}}(u,v)$. We now show two properties: (1) the mincut gains relatively little weight from $\widetilde{H}$ in the union H', and (2) any balanced cut automatically has at least λ total weight from $\widetilde{H}$.

1. For a mincut $\partial_G S^*$ in G with $\mathbf{vol}_G(S^*) \le |\partial_G S^*|/\phi = \lambda/\phi$, the cut crosses
$$w(\partial_{\widehat{H}} S^*) \le \mathbf{vol}_{\widehat{H}}(S^*) \le \Theta(1) \cdot \mathbf{vol}_G(S^*)/\lambda \le \Theta(1/\phi)$$
edges in $\widetilde{H}$, for a total cost of at most $\Theta(1/\phi) \cdot \Theta(\epsilon\phi\lambda) \le \epsilon\lambda$.
2. For a balanced cut $\partial_G S$, it satisfies $|\partial_G S| \ge \phi \cdot \mathbf{vol}_G(S) \ge \alpha\lambda$, so it crosses
$$w(\partial_{\widehat{H}} S) \ge \Theta(1) \cdot \mathbf{vol}_{\widehat{H}}(S) \ge \Theta(1) \cdot \mathbf{vol}_G(S)/\lambda \ge \Theta(\alpha/\phi)$$
many edges in $\widetilde{H}$, for a total cost of at least $\Theta(\alpha/\phi) \cdot \Theta(\epsilon\phi\lambda)$. Setting $\alpha := \Theta(\frac{1}{\epsilon})$, the cost becomes at least λ.

Therefore, in the weighted graph H', the mincut has weight at most $(1+O(\epsilon))\lambda$, and any cut has weight at least $(1-\epsilon)\lambda$. We can reset ϵ to be a constant factor smaller so that the factor $(1+O(\epsilon))$ becomes $(1+\epsilon)$.

To finish the proof of Theorem 6.4.1, it remains to extract an unweighted graph H and a weight W from the weighted graph H'. Since $\widehat{W} = \Theta(\frac{\epsilon^2\phi\lambda}{\alpha \log n}) = \Theta(\frac{\epsilon^3\phi\lambda}{\log n})$ and $\widetilde{W} = \Theta(\epsilon\phi\lambda)$, we can make $\widetilde{W}$ an integer multiple of $\widehat{W}$, so that each edge in H' is an integer multiple of $\widehat{W}$. We can therefore set $W := \widehat{W}$ and define the unweighted graph H so that $\#_H(u,v) = w_{H'}(u,v)/\widehat{W}$ for all $u,v \in V$.

6.5 General Case

This section is dedicated to proving Theorem 6.3.4. For simplicity, we instead prove the following restricted version first, which has the additional assumption that the maximum edge weight in G is bounded. At the end of this section, we show why this assumption can be removed to obtain the full Theorem 6.3.4.

Theorem 6.5.1: Sparsifier with maximum weight assumption

There exists a function $f(n) \le 2^{O(\log n)^{5/6}(\log\log n)^{O(1)}}$ such that the following holds. Let G be a graph with mincut λ **and maximum edge weight at most** $\epsilon^4\lambda/f(n)$**.** For any $0 < \epsilon \le 1$, we can compute, in deterministic $2^{O(\log n)^{5/6}(\log\log n)^{O(1)}} m$ time, an unweighted graph H and some weight $W \ge \epsilon^4\lambda/f(n)$ such that the two properties of Theorem 6.3.4 hold, i.e.,

1. For any mincut S^* of G, we have $W \cdot |\partial_H S^*| \le (1+\epsilon)\lambda$, and
2. For any cut $\emptyset \subsetneq S \subsetneq V$ of G, we have $W \cdot |\partial_H S| \ge (1-\epsilon)\lambda$.

6.5.1 Expander Decomposition Preliminaries

Our main tool in generalizing the expander case is *expander decompositions*, which was popularized by Spielman and Teng [100] and is quickly gaining traction in the area of fast graph algorithms. The general approach to utilizing expander decompositions is as follows. First, solve the case when the input graph is an expander, which we have done in Section 6.4 for the problem described in Theorem 6.3.4. Then, for a general graph, *decompose* it into a collection of expanders with few edges between the expanders, solve the problem each expander separately, and combine the solutions together, which often involves a recursive call on a graph that is a constant-factor smaller. For our purposes, we use a slightly stronger variant than the usual expander decomposition that ensures *boundary-linkedness*, which will be important in our analysis. The following definition is inspired by [45]; note that our variant is weaker than the one in Definition 4.2 of [45] in that we only guarantee their property (2). For completeness, we include a full proof in Section 8.8 that is similar to the one in [45].

Theorem 6.5.2: Boundary-linked expander decomposition, restated

Let $G = (V, E)$ be a graph and let $r \geq 1$ be a parameter. There is a deterministic algorithm in $m^{1+O(1/r)} + \tilde{O}(m/\phi^2)$ time that, for any parameters $\beta \leq (\log n)^{-O(r^4)}$ and $\phi \leq \beta$, partitions $V = V_1 \uplus \cdots \uplus V_k$ such that

1. Each vertex set V_i satisfies

$$\min_{\emptyset \subsetneq S \subsetneq V_i} \frac{w(\partial_{G[V_i]} S)}{\min\{\mathbf{vol}_{G[V_i]}(S) + \frac{\beta}{\phi} w(E_G(S, V \setminus V_i)), \mathbf{vol}_{G[V_i]}(V_i \setminus S) + \frac{\beta}{\phi} w(E_G(V_i \setminus S, V \setminus V_i))\}} \geq \phi. \tag{6.4}$$

   Informally, we call the graph $G[V_i]$ together with its boundary edges $E_G(V_i, V \setminus V_i)$ a *β-boundary-linked* ϕ-expander.In particular, for any S satisfying

$$\mathbf{vol}_{G[V_i]}(S) + \frac{\beta}{\phi} w(E_G(S, V \setminus V_i)) \leq \mathbf{vol}_{G[V_i]}(V_i \setminus S) + \frac{\beta}{\phi} w(E_G(V_i \setminus S, V \setminus V_i)),$$

   we simultaneously obtain

$$\frac{w(\partial_{G[V_i]} S)}{\mathbf{vol}_{G[V_i]}(S)} \geq \phi \quad \text{and} \quad \frac{w(\partial_{G[V_i]} S)}{\frac{\beta}{\phi} w(E_G(S, V \setminus V_i))} \geq \phi \iff \frac{w(\partial_{G[V_i]} S)}{w(E_G(S, V \setminus V_i))} \geq \beta.$$

   The right-most inequality is where the name "boundary-linked" comes from.

2. The total weight of "inter-cluster" edges, $w(\partial V_1 \cup \cdots \cup \partial V_k)$, is at most $(\log n)^{O(r^4)} \phi \mathbf{vol}(V)$.

Note that for our applications, it's important that the boundary-linked parameter β is much larger than ϕ. This is because in our recursive algorithm, the approximation factor will blow up by roughly $1/\beta$ per recursion level, while the instance size shrinks by roughly

ϕ.

In order to capture recursion via expander decompositions, we now define a *boundary-linked expander decomposition sequence* $\{G^i\}$ on the graph G in a similar way to [45]. Compute a boundary-linked expander decomposition for β and $\phi \leq \beta$ to be determined later, contract each expander,[1] and recursively decompose the contracted graph until the graph consists of a single vertex. Let $G^0 = G$ be the original graph and $G^1, G^2, \dots, G^L$ be the recursive contracted graphs. Note that each graph G^i has minimum degree at least λ, since any degree cut in any G^i induces a cut in the original graph G. Each time we contract, we will keep edge identities for the edges that survive, so that $E(G^0) \supseteq E(G^1) \supseteq \cdots \supseteq E(G^L)$. Let U^i be the vertices of G^i.

For the rest of Section 6.5.1, fix an expander decomposition sequence $\{G^i\}$ of G. For any subset $\emptyset \subsetneq S \subsetneq V$, we now define an *decomposition sequence* of S as follows. Let $S^0 = S$, and for each $i > 0$, construct S^{i+1} as a subset of the vertices of G^{i+1}, as follows. Take the expander decomposition of G^i, which partitions the vertices U^i of G^i into, say, $U^i_1, \dots, U^i_{k_i}$. Each of the U^i_j gets contracted to a single vertex u_j in G^i. For each U^i_j, we have a choice whether to add u_j to S^i or not. This completes the construction of S^i. Define the "difference" $D^i_j = U_j \setminus S^i$ if $u_j \in S^i$, and $D^i_j = U_j \cap S^i$ otherwise. The sets S^i, U^i_j, and D^i_j define the decomposition sequence of S.

We now prove some key properties of the boundary-linked expander decomposition sequence in the context of graph cuts, which we will use later on. First, regardless of the choice whether to add each u_j to S^i, we have the following lemma relating the sets D^i_j to the original set S.

Lemma 6.5.3

For any decomposition sequence $\{S^i\}$ of S,

$$\partial_G S \subseteq \bigcup_{i=0}^{L} \bigcup_{j \in [k_i]} \partial_{G^i} D^i_j.$$

Proof. Observe that

$$(\partial_{G^i} S^i) \triangle (\partial_{G^{i+1}} S^{i+1}) \subseteq \bigcup_{j \in [k_i]} \partial_{G^i} D^i_j. \tag{6.5}$$

In particular,

$$\partial_{G^i} S^i \subseteq \partial_{G^{i+1}} S^{i+1} \cup \bigcup_{j \in [k_i]} \partial_{G^i} D^i_j.$$

[1]Since we are working with weighted multigraphs, we do *not* collapse parallel edges obtained from contraction into single edges.

Iterating this over all i,

$$\partial_G S \subseteq \bigcup_{i=0}^{L} \bigcup_{j\in[k_i]} \partial_{G^i} D_j^i.$$

□

We now define a specific decomposition sequence of S, by setting up the rule whether or not to include each u_j in S^i. For each U_j^i, if

$$\mathbf{vol}_{G^i[U_j^i]}(S^i \cap U_j^i) + \frac{\beta}{\phi} w(E_{G^i}(S^i \cap U_j^i, U^i \setminus U_j^i)) \geq \mathbf{vol}_{G^i[U_j^i]}(U_j^i \setminus S^i) + \frac{\beta}{\phi} w(E_{G^i}(U_j^i \setminus S^i, U^i \setminus U_j^i)),$$

then add u_j to S^i; otherwise, do not add u_j to S^i. This ensures that

$$\mathbf{vol}_{G^i[U_j^i]}(U_j^i \setminus D_j^i) + \frac{\beta}{\phi} w(E_{G^i}(U_j^i \setminus D_j^i, U^i \setminus U_j^i)) \geq \mathbf{vol}_{G^i[U_j^i]}(D_j^i) + \frac{\beta}{\phi} w(E_{G^i}(D_j^i, U^i \setminus U_j^i)). \tag{6.6}$$

Since $G^i[U_j^i]$ is a β-boundary-linked ϕ-expander, by our construction, we have, for all i, j,

$$\frac{w(\partial_{G^i[U_j^i]} D_j^i)}{\mathbf{vol}_{G^i[U_j^i]}(D_j^i)} \geq \phi \tag{6.7}$$

and

$$\frac{w(\partial_{G^i[U_j^i]} D_j^i)}{w(E_{G^i}(D_j^i, U^i \setminus U_j^i))} \geq \beta. \tag{6.8}$$

For this specific construction of $\{S^i\}$, called the *canonical* decomposition sequence of S, we have the following lemma, which complements Lemma 6.5.3.

Lemma 6.5.4

Let $\{S^i\}$ be any decomposition sequence of S satisfying (6.8) for all i, j. Then,

$$\sum_{i=0}^{L} \sum_{j\in[k_i]} w(\partial_{G^i} D_j^i) \leq \beta^{-O(L)} w(\partial_G S).$$

Proof. By (6.8),

$$w(E_{G^i}(D_j^i, U^i \setminus U_j^i)) \leq \frac{1}{\beta} \cdot w(\partial_{G^i[U_j^i]} D_j^i).$$

The edges of $\partial_{G^i[U^i_j]} D^i_j$ are inside $\partial_{G^i} S^i$ and are disjoint over distinct j, so in total,

$$\sum_{j\in[k_i]} w(\partial_{G^i} D^i_j) \le \sum_{j\in[k_i]} \frac{1}{\beta}\cdot w(\partial_{G^i[U^i_j]} D^i_j) \le \frac{1}{\beta}\cdot w(\partial_{G^i} S^i).$$

From (6.5), we also obtain

$$\partial_{G^{i+1}} S^{i+1} \subseteq \partial_{G^i} S^i \cup \bigcup_{j\in[k_i]} \partial_{G^i} D^i_j.$$

Therefore,

$$w(\partial_{G^{i+1}} S^{i+1}) \le w(\partial_{G^i} S^i) + w\left(\bigcup_{j\in[k_i]} \partial_{G^i} D^i_j\right) \le \left(1+\frac{1}{\beta}\right)\cdot w(\partial_{G^i} S^i).$$

Iterating this over all $i \in [L]$, we obtain

$$w(\partial_{G^i} S^i) \le \left(1+\frac{1}{\beta}\right)^i \cdot w(\partial_G S).$$

Thus,

$$\sum_{i=0}^{L}\sum_{j\in[k_i]} w(\partial_{G^i} D^i_j) \le \sum_{i=0}^{L} \frac{1}{\beta}\cdot w(\partial_{G^i} S^i) \le \sum_{i=0}^{L} \frac{1}{\beta}\cdot\left(1+\frac{1}{\beta}\right)^i \cdot w(\partial_G S) = \beta^{-O(L)} w(\partial_G S).$$

□

6.5.2 Unbalanced Case

In this section, we generalize the notion of *unbalanced* from Section 6.4 to the general case, and then prove a $(1+\epsilon)$-approximate sparsifier of the unbalanced cuts.

Fix an expander decomposition sequence $\{G^i\}$ of G for the Section 6.5.2. For a given set $\emptyset \subsetneq S \subsetneq V$, let $\{S^i\}$ be the canonical decomposition sequence of S, and define D^i_j as before, so that they satisfy (6.7) and (6.8) for all i, j. We generalize our definition of *unbalanced* from the expander case as follows, for some $\tau = n^{o(1)}$ to be specified later.

Definition 6.5.5

The set $S \subseteq V$ is *τ-unbalanced* if for each level i, $\sum_{j\in[k_i]} \mathbf{vol}_{G^i}(D^i_j) \le \tau\lambda/\phi$. A cut ∂S is τ-unbalanced if the set S is τ-unbalanced.

Note that if G is originally an expander, then in the first expander decomposition of the sequence, we can declare the entire graph as a single expander; in this case, the expander decomposition sequence stops immediately, and the definition of τ-unbalanced becomes equivalent to that from the expander case. We now claim that for an appropriate value of τ, any mincut is τ-unbalanced.

Claim 6.5.6

For $\tau \geq \beta^{-\Omega(L)}$, any mincut ∂S^* of G is τ-unbalanced.

Proof. Consider the canonical decomposition sequence of S, and define D^i_j as usual. For each level i and index $j \in [k_i]$,

$$\begin{aligned}\mathbf{vol}_{G^i}(D^i_j) &= \mathbf{vol}_{G^i[U^i_j]}(D^i_j) + w(E_{G^i}(D^i_j, U^i \setminus U^i_j)) \\ &\overset{(6.7)}{\leq} \frac{1}{\phi} w(\partial_{G^i[U^i_j]} D^i_j) + w(E_{G^i}(D^i_j, U^i \setminus U^i_j)) \\ &\leq \frac{1}{\phi} w(\partial_{G^i} D^i_j).\end{aligned}$$

Summing over all $j \in [k_i]$ and applying Lemma 6.5.4,

$$\sum_{j\in[k_i]} \mathbf{vol}_{G^i}(D^i_j) \leq \sum_{j\in[k_i]} \frac{1}{\phi} w(\partial_{G^i} D^i_j) = \frac{1}{\phi} \cdot \sum_{j\in[k_i]} w(\partial_{G^i} D^i_j) \overset{\text{Lem.6.5.4}}{\leq} \frac{1}{\phi} \cdot \beta^{-O(L)} w(\partial_G S^*) \leq \frac{\tau\lambda}{\phi},$$

so S^* is τ-unbalanced. □

Let us now introduce some notation exclusive to this section. For each vertex $v \in U^i$, let $\overline{v} \subseteq V$ be its "pullback" on the original set V, defined as all vertices in V that get contracted into v in graph G^i in the expander sequence. For each set D^i_j, let $\overline{D^i_j} \subseteq V$ be the pullback of D^i_j, defined as $\overline{D^i_j} = \bigcup_{v\in D^i_j} \overline{v}$. We can then write

$$\mathbb{1}_S = \sum_{i,j} \pm \mathbb{1}_{\overline{D^i_j}} = \sum_{i,j} \sum_{v \in D^i_j} \pm \mathbb{1}_{\overline{v}},$$

where the $\pm$ sign depends on whether $D^i_j = U^i_j \setminus S^i$ or $D^i_j = U^i_j \cap S^i$. Then,

$$w(\partial_G S) = \mathbb{1}_S^T L_G \mathbb{1}_S = \sum_{i,j,k,l} \pm \mathbb{1}^T_{\overline{D^i_j}} L_G \mathbb{1}_{\overline{D^k_l}} = \sum_{i,j,k,l} \sum_{u\in D^i_j, v\in D^k_l} \pm \mathbb{1}^T_{\overline{u}} L_G \mathbb{1}_{\overline{v}}. \tag{6.9}$$

Claim 6.5.7

For an τ-unbalanced set S, there are at most $((L+1)\tau/\phi)^2$ nonzero terms in the summation (6.9).

Proof. Each vertex $v \in D^i_j$ has degree at least λ in G^i, since it induces a cut (specifically, its pullback $\overline{v} \subseteq V$) in the original graph G. Therefore,

$$\tau\lambda/\phi \geq \sum_{j \in [k_i]} \mathbf{vol}_{G^i}(D^i_j) \geq \sum_{j \in [k_i]} |D^i_j| \cdot \lambda,$$

so there are at most τ/ϕ many choices for j and $u \in D^i_j$ given a level i. There are at most $L+1$ many choices for i, giving at most $(L+1)\tau/\phi$ many combinations of i, j, u. The same holds for combinations of k, l, v, hence the claim. □

The main goal of this section is to prove the following lemma.

Lemma 6.5.8

There exists a constant $C > 0$ such that given any weight $W \leq \frac{C\epsilon\phi\lambda}{\tau \ln(Lm)}$, we can compute, in deterministic $\tilde{O}(L^2 m)$ time,[a] an unweighted graph H such that for all levels i, k and vertices $u \in U^i, v \in U^k$ satisfying $\deg_{G^i}(u) \leq \tau\lambda/\phi$ and $\deg_{G^k}(v) \leq \tau\lambda/\phi$,

$$\left| \mathbb{1}^T_{\overline{u}} L_G \mathbb{1}_{\overline{v}} - W \cdot \mathbb{1}^T_{\overline{u}} L_H \mathbb{1}_{\overline{v}} \right| \leq \epsilon\lambda. \tag{6.10}$$

[a]outside of computing the boundary-linked expander decomposition sequence

Before we prove Lemma 6.5.8, we show that it implies a sparsifier of τ-unbalanced cuts, which is the lemma we will eventually use to prove Theorem 6.5.1:

Lemma 6.5.9

There exists a constant $C > 0$ such that given any weight $W \leq \frac{C\epsilon\phi\lambda}{\tau \ln(Lm)}$, we can compute, in deterministic $\tilde{O}(L^2 m)$ time, an unweighted graph H such that for each *τ-unbalanced* cut S,

$$\left| w(\partial_G S) - W \cdot w(\partial_H S) \right| \leq \left(\frac{(L+1)\tau}{\phi} \right)^2 \cdot \epsilon\lambda.$$

Proof. Let $C > 0$ be the same constant as the one in Lemma 6.5.8. Applying (6.9) to $\partial_H S$ as well, we have

$$w(\partial_G S) - W \cdot w(\partial_H S) = \sum_{i,j,k,l} \sum_{u \in D^i_j, v \in D^k_l} \pm(\mathbb{1}^T_{\overline{u}} L_G \mathbb{1}_{\overline{v}} - W \cdot \mathbb{1}^T_{\overline{u}} L_H \mathbb{1}_{\overline{v}}),$$

so that

$$\left| w(\partial_G S) - W \cdot w(\partial_H S) \right| \leq \sum_{i,j,k,l} \sum_{u \in D^i_j, v \in D^k_l} \left| \mathbb{1}^T_{\overline{u}} L_G \mathbb{1}_{\overline{v}} - W \cdot \mathbb{1}^T_{\overline{u}} L_H \mathbb{1}_{\overline{v}} \right|.$$

By Claim 6.5.7, there are at most $((L+1)\tau/\phi)^2$ nonzero terms in the summation above. In order to apply Lemma 6.5.8 to each such term, we need to show that $\deg_{G^i}(u) \le \tau\lambda/\phi$ and $\deg_{G^k}(v) \le \tau\lambda/\phi$. Since S is an τ-unbalanced cut, we have

$$\deg_{G^i}(u) \le \mathbf{vol}_{G^i}(D^i_j) \le \sum_{j\in[k_i]} \mathbf{vol}_{G^i}(D^i_j) \le \tau\lambda/\phi,$$

and similarly for $\deg_{G^k}(v)$. Therefore, by Lemma 6.5.8,

$$\left|w(\partial_G S) - W \cdot w(\partial_H S)\right| \le \left(\frac{(L+1)\tau}{\phi}\right)^2 \cdot \epsilon\lambda,$$

as desired. □

The rest of Section 6.5.2 is dedicated to proving Lemma 6.5.8.

Expand out $L_G = \sum_{e\in E} L_e$, where L_e is the Laplacian of the graph consisting of the single edge e of the same weight, so that $\mathbb{1}_{\overline{u}}^T L_e \mathbb{1}_{\overline{v}} \in \{-w(e), w(e)\}$ if exactly one endpoint of e is in $\overline{u}$ and exactly one endpoint of e is in $\overline{v}$, and $\mathbb{1}_{\overline{u}}^T L_e \mathbb{1}_{\overline{v}} = 0$ otherwise. Let $E_{\overline{u},\overline{v},+}$ denote the edges $e \in E$ with $\mathbb{1}_{\overline{u}}^T L_e \mathbb{1}_{\overline{v}} = w(e)$, and $E_{\overline{u},\overline{v},-}$ denote those with $\mathbb{1}_{\overline{u}}^T L_e \mathbb{1}_{\overline{v}} = -w(e)$.

Random Sampling Procedure Consider the Benzcur-Karger random sampling procedure, which we will de-randomize in this section. Let $\widehat{H}$ be a subgraph of G with each edge $e \in E$ sampled independently with probability $w(e)/W$, which is at most 1 by the assumption of Theorem 6.5.1. Intuitively, the parameter $W \ge \lambda/f(n)$ is selected so that with probability close to 1, (6.10) holds over all i, k, u, v.

We now introduce our concentration bounds for the random sampling procedure, namely the classical multiplicative Chernoff bound. We state a form that includes bounds on the moment-generating function $\mathbb{E}[e^{tX}]$ obtained in the standard proof.

Lemma 6.5.10: Multiplicative Chernoff bound

Let $X_1, \ldots, X_N$ be independent random variables that take values in $[0, 1]$, and let $X = \sum_{i=1}^N X_i$ and $\mu = \mathbb{E}[X] = \sum_{i=1}^N p_i$. Fix a parameter δ, and define

$$t^u = \ln(1+\delta) \qquad \text{and} \qquad t^l = \ln\left(\frac{1}{1-\delta}\right). \tag{6.11}$$

Then, we have the following upper and lower tail bounds:

$$\Pr[X > (1+\delta)\mu] \le e^{-t^u(1+\delta)\mu}\mathbb{E}[e^{t^u X}] \le e^{-\delta^2\mu/3}, \tag{6.12}$$
$$\Pr[X < (1-\delta)\mu] \le e^{t^l(1-\delta)\mu}\mathbb{E}[e^{-t^l X}] \le e^{-\delta^2\mu/3}. \tag{6.13}$$

We now describe our de-randomization by pessimistic estimators. Let $F \subseteq E$ be the set of edges for which a value $X_e \in \{0, 1\}$ has already been set, so that F is initially $\emptyset$. For each i, k, vertices $u \in U^i, v \in U^k$, and sign $\circ \in \{+, -\}$ such that $E_{\overline{u},\overline{v},\circ} \neq \emptyset$, we first define a "local" pessimistic estimator $\Phi_{\overline{u},\overline{v},\circ}(\cdot)$, which is a function on the set of pairs (e, X_e) over all $e \in F$. The algorithm computes a 3-approximation $\widetilde{\lambda} \in [\lambda, 3\lambda]$ to the mincut with the $\tilde{O}(m)$-time $(2+\epsilon)$-approximation algorithm of Matula [80], and sets

$$\mu_{\overline{u},\overline{v},\circ} = \frac{w(E_{\overline{u},\overline{v},\circ})}{W} \qquad \text{and} \qquad \delta_{\overline{u},\overline{v},\circ} = \frac{\epsilon\widetilde{\lambda}}{6w(E_{\overline{u},\overline{v},\circ})}. \tag{6.14}$$

Following (6.11), we define

$$t^u_{\overline{u},\overline{v},\circ} = \ln(1 + \delta_{\overline{u},\overline{v},\circ}) \qquad \text{and} \qquad t^l_{\overline{u},\overline{v},\circ} = \ln\left(\frac{1}{1 - \delta_{\overline{u},\overline{v},\circ}}\right), \tag{6.15}$$

and following the middle expressions (the moment-generating functions) in (6.12) and (6.13), we define

$$\begin{aligned}\Phi_{\overline{u},\overline{v},\circ}(\{(e, X_e) : e \in F\}) = e^{-t^u_{\overline{u},\overline{v},\circ}(1+\delta_{\overline{u},\overline{v},\circ})\mu_{\overline{u},\overline{v},\circ}} &\prod_{e \in E_{\overline{u},\overline{v},\circ} \cap F} e^{t^u_{\overline{u},\overline{v},\circ} X_e} \prod_{e \in E_{\overline{u},\overline{v},\circ} \setminus F} \mathbb{E}[e^{t^u_{\overline{u},\overline{v},\circ} X_e}] \\ + e^{t^l_{\overline{u},\overline{v},\circ}(1-\delta_{\overline{u},\overline{v},\circ})\mu_{\overline{u},\overline{v},\circ}} &\prod_{e \in E_{\overline{u},\overline{v},\circ} \cap F} e^{-t^l_{\overline{u},\overline{v},\circ} X_e} \prod_{e \in E_{\overline{u},\overline{v},\circ} \setminus F} \mathbb{E}[e^{-t^l_{\overline{u},\overline{v},\circ} X_e}].\end{aligned}$$

Observe that if we are setting the value of $X_{e'}$ for a new edge $e' \in E_{\overline{u},\overline{v},\circ} \setminus F$, then by linearity of expectation, there is an assignment $X_{e'} \in \{0, 1\}$ for which $\Phi_{\overline{u},\overline{v},\circ}(\cdot)$ does not decrease:

$$\Phi_{\overline{u},\overline{v},\circ}(\{(e, X_e) : e \in F\} \cup (e', X_{e'})) \leq \Phi_{\overline{u},\overline{v},\circ}(\{(e, X_e) : e \in F\}).$$

Since the X_e terms are independent, we have that for any $t \in \mathbb{R}$ and $E' \subseteq E$,

$$\mathbb{E}\left[e^{t \sum_{e \in E'} X_e}\right] = \prod_{e \in E'} \mathbb{E}[e^{tX_e}].$$

By the independence above and the second inequalities in (6.12) and (6.13), the initial "local" pessimistic estimator $\Phi_{\overline{u},\overline{v},\circ}(\emptyset)$ satisfies

$$\begin{aligned}\Phi_{\overline{u},\overline{v},\circ}(\emptyset) \leq 2\exp\left(-\frac{\delta^2_{\overline{u},\overline{v},\circ}\mu_{\overline{u},\overline{v},\circ}}{3}\right) &= 2\exp\left(-\frac{(\epsilon\widetilde{\lambda}/(6w(E_{\overline{u},\overline{v},\circ})))^2 \cdot w(E_{\overline{u},\overline{v},\circ})/W \cdot}{3}\right) \\ &= 2\exp\left(-\frac{\epsilon\widetilde{\lambda}^2}{108w(E_{\overline{u},\overline{v},\circ})W}\right).\end{aligned}$$

We would like the above expression to be less than 1. To upper bound $w(E_{\overline{u},\overline{v},\circ})$, note first that every edge $e \in E_{\overline{u},\overline{v},\circ}$ must, under the contraction from G all the way to G^i, map to an edge incident to u in G^i, which gives $w(E_{\overline{u},\overline{v},\circ}) \leq \deg_{G^i}(u)$. Moreover, since $\deg_{G^i}(u) \leq \tau\lambda/\phi$

by assumption, we have

$$w(E_{\overline{u},\overline{v},\circ}) \leq \deg_{G^i}(u) \leq \tau\lambda/\phi \tag{6.16}$$

so that

$$\Phi_{\overline{u},\overline{v},\circ}(\emptyset) \leq 2\exp\left(-\frac{\epsilon\widetilde{\lambda}^2}{108(\tau\lambda/\phi)W}\right) \leq 2\exp\left(-\frac{\epsilon\lambda^2}{108(\tau\lambda/\phi)W}\right) = 2\exp\left(-\frac{\epsilon\phi\lambda}{108\tau W}\right).$$

Assume that

$$W \leq \frac{\epsilon\phi\lambda}{108\tau\ln\left(16(L+1)^2m\right)}, \tag{6.17}$$

which satisfies the bounds in Lemma 6.5.8, so that

$$\Phi_{\overline{u},\overline{v},\circ}(\emptyset) \leq 2\exp\left(-\frac{\epsilon\phi\lambda}{108\tau W}\right) \leq \frac{1}{8(L+1)^2m}.$$

Our actual, "global" pessimistic estimator $\Phi(\cdot)$ is simply the sum of the "local" pessimistic estimators:

$$\Phi(\{(e,X_e) : e \in F\}) = \sum_{\substack{i,k,\\ u\in U^i, v\in U^k,\\ \circ\in\{+,-\}}} \Phi_{\overline{u},\overline{v},\circ}(\{(e,X_e) : e \in F\}).$$

The initial pessimistic estimator $\Phi(\emptyset)$ satisfies

$$\Phi(\emptyset) = \sum_{\substack{i,k,\\ u\in U^i, v\in U^k,\\ \circ\in\{+,-\}}} \Phi_{\overline{u},\overline{v},\circ}(\emptyset) \leq \sum_{\substack{i,k,\\ u\in U^i, v\in U^k,\\ \circ\in\{+,-\}}} \frac{1}{8(L+1)^2m} \overset{\text{Clm.6.5.12}}{\leq} 4(L+1)^2m \cdot \frac{1}{8(L+1)^2m} = \frac{1}{2}.$$

Again, if we are setting the value of X_f for a new edge $f \in E \setminus F$, then by linearity of expectation, there is an assignment $X_f \in \{0,1\}$ for which $\Phi(\cdot)$ does not decrease:

$$\Phi(\{(e,X_e) : e \in F\} \cup (f,X_f)) \leq \Phi(\{(e,X_e) : e \in F\}).$$

Therefore, if we always select such an assignment X_e, then once we have iterated over all $e \in E$, we have

$$\Phi(\{(e,X_e) : e \in E\}) \leq \Phi(\emptyset) \leq \frac{1}{2} \leq 1. \tag{6.18}$$

This means that for each $i, k, u \in U^i, v \in U^k$, and sign $\circ \in \{+, -\}$,

$$\begin{aligned}&\Phi_{\overline{u},\overline{v},\circ}(\{(e, X_e) : e \in E\}) \\ &\quad = e^{-t^u_{\overline{u},\overline{v},\circ}(1+\delta_{\overline{u},\overline{v},\circ})\mu_{\overline{u},\overline{v},\circ}} \prod_{e\in E_{\overline{u},\overline{v},\circ}} e^{t^u_{\overline{u},\overline{v},\circ}X_e} + e^{t^l_{\overline{u},\overline{v},\circ}(1-\delta_{\overline{u},\overline{v},\circ})\mu_{\overline{u},\overline{v},\circ}} \prod_{e\in E_{\overline{u},\overline{v},\circ}} e^{-t^l_{\overline{u},\overline{v},\circ}X_e} \le 1.\end{aligned}$$

In particular, each of the two terms is at most 1. Recalling from definition (6.14) that $\mu_{\overline{u},\overline{v},\circ} = w(E_{\overline{u},\overline{v},\circ})/W$ and $\delta_{\overline{u},\overline{v},\circ} = \epsilon\widetilde{\lambda}/(6w(E_{\overline{u},\overline{v},\circ}))$, we have

$$\sum_{e\in E_{\overline{u},\overline{v},\circ}} X_e \le (1 + \delta_{\overline{u},\overline{v},\circ})\mu_{\overline{u},\overline{v},\circ} = \frac{w(E_{\overline{u},\overline{v},\circ})}{W} + \frac{\epsilon\widetilde{\lambda}}{6W}$$

and

$$\sum_{e\in E_{\overline{u},\overline{v},\circ}} X_e \ge (1 - \delta_{\overline{u},\overline{v},\circ})\mu_{\overline{u},\overline{v},\circ} = \frac{w(E_{\overline{u},\overline{v},\circ})}{W} - \frac{\epsilon\widetilde{\lambda}}{6W}.$$

Therefore,

$$\left|\mathbb{1}^T_{\overline{u}} L_G \mathbb{1}_{\overline{v}} - W \cdot \mathbb{1}^T_{\overline{u}} L_{\widehat{H}} \mathbb{1}_{\overline{v}}\right| \le \sum_{\circ\in\{+,-\}} \left| w(E_{\overline{u},\overline{v},\circ}) - W \cdot \sum_{e\in E_{\overline{u},\overline{v},\circ}} X_e \right| \le \frac{\epsilon\widetilde{\lambda}}{6} + \frac{\epsilon\widetilde{\lambda}}{6} = \frac{\epsilon\widetilde{\lambda}}{3} \le \epsilon\lambda,$$

fulfilling (6.10).

It remains to consider the running time. We first bound the number of i, k, u, v such that either $E_{\overline{u},\overline{v},+} \ne \emptyset$ or $E_{\overline{u},\overline{v},-} \ne \emptyset$; the others are irrelevant since $\mathbb{1}^T_{\overline{u}} L_G \mathbb{1}_{\overline{v}} = \mathbb{1}^T_{\overline{u}} L_{\widehat{H}} \mathbb{1}_{\overline{v}} = 0$.

Claim 6.5.11

For each pair of vertices x, y, there are at most $(L + 1)^2$ many selections of i, k and $u \in U^i, v \in U^k$ such that $x \in \overline{u}$ and $y \in \overline{v}$.

Proof. For each level i, there is exactly one vertex $u \in U^i$ with $x \in \overline{u}$, and for each level k, there is exactly one vertex $v \in U^k$ with $y \in \overline{v}$. This makes $(L + 1)^2$ many choices of i, k total, and unique choices for u, v given i, k. □

Claim 6.5.12

For each edge $e \in E$, there are at most $4(L+1)^2$ many selections of i, k and $u \in U^i, v \in U^k$ such that $e \in E_{\overline{u},\overline{v},+} \cup E_{\overline{u},\overline{v},-}$.

Proof. If $e \in E_{\overline{u},\overline{v},+} \cup E_{\overline{u},\overline{v},-}$, then exactly one endpoint of e is in $\overline{u}$ and exactly one endpoint of e is in $\overline{v}$. There are four possibilities as to which endpoint is in $\overline{u}$ and which is in $\overline{v}$, and for each, Claim 6.5.11 gives at most $(L + 1)^2$ choices. □

Claim 6.5.13

There are at most $4(L+1)^2m$ many choices of i, k, u, v such that either $E_{\overline{u},\overline{v},+} \neq \emptyset$ or $E_{\overline{u},\overline{v},-} \neq \emptyset$.

Proof. For each such choice, charge it to an arbitrary edge $(x, y) \in E_{\overline{u},\overline{v},+} \cup E_{\overline{u},\overline{v},-}$. Each edge is charged at most $4(L+1)^2$ times by Claim 6.5.12, giving at most $4(L+1)^2m$ total charges. □

By Claim 6.5.12, each new edge $e \in E \setminus F$ is in at most $4(L+1)^2$ many sets $E_{\overline{u},\overline{v},\circ}$, and therefore affects at most $4(L+1)^2$ many terms $\Phi_{\overline{u},\overline{v},\circ}(\{(e, X_e) : e \in F\})$. The algorithm only needs to re-evaluate these terms with the new variable X_e set to 0 and with it set to 1, and take the one with the smaller new $\Phi(\cdot)$. This takes $O(L^2)$ arithmetic operations.

How long do the arithmetic operations take? We compute each exponential in $\Phi(\cdot)$ with $c \log n$ bits of precision after the decimal point for some constant $c > 0$, which takes polylog(n) time. Each one introduces an additive error of $1/n^c$, and there are poly(n) exponential computations overall, for a total of $1/n^c \cdot \text{poly}(n) \leq 1/2$ error for a large enough $c > 0$. Factoring in this error, the inequality (6.18) instead becomes

$$\Phi(\{(e, X_e) : e \in E\}) \leq \Phi(\emptyset) + \frac{1}{2} \leq \frac{1}{2} + \frac{1}{2} = 1,$$

so the rest of the bounds still hold.

This concludes the proof of Lemma 6.5.8.

6.5.3 Balanced Case

Similar to the expander case, we treat balanced cuts by "overlaying" a "lossy", $n^{o(1)}$-approximate sparsifier of G top of the graph $\widehat{H}$ obtained from Lemma 6.5.9. In the expander case, this sparsifier was just another expander, but for general graphs, we need to do more work. At a high level, we compute an expander decomposition sequence, and on each level, we replace each of the expanders with a fixed expander (like in the expander case).

Theorem 6.5.14: Lossy cut sparsifier

Let G be an weighted multigraph with mincut λ whose edges have weight at most $O(\lambda)$. For any parameters $\widetilde{\lambda} \in [\lambda, 3\lambda]$ and $\Delta \geq 2^{O(\log n)^{5/6}}$, we can compute, in deterministic $2^{O(\log n)^{5/6}(\log \log n)^{O(1)}}m + O(\Delta m)$ time, an unweighted multigraph H such that $W \cdot H$ is a γ-approximate cut sparsifier of G, where $\gamma \leq 2^{O(\log n)^{5/6}(\log \log n)^{O(1)}}$ and $W = \widetilde{\lambda}/\Delta$. (The graph H does not need to be a subgraph of G.) Moreover, the algorithm does not need to know the mincut value λ.

6.5.4 Combining Them Together

We now combine the unbalanced and balanced cases to prove Theorem 6.5.1. Our high-level procedure is similar to the one from the expander case. For the τ-unbalanced cuts, we use Lemma 6.5.9. For the balanced cuts, we show that their size must be much larger than λ, so that even on a γ-approximate weighted sparsifier guaranteed by Theorem 6.5.14, their weight is still much larger than λ. We then "overlay" the γ-approximate weighted sparsifier with a "light" enough weight onto the sparsifier of τ-unbalanced cuts. The weight is light enough to barely affect the mincuts, but still large enough to force any balanced cut to increase by at least λ in weight.

Claim 6.5.15

If a cut S is balanced, then $w(\partial_G S) \geq \beta^{O(L)}\tau\lambda$.

Proof. Consider the level i for which $\sum_{j\in[k_i]} \mathbf{vol}_{G^i}(D^i_j) > \tau\lambda/\phi$. For each $j \in [k_i]$, we have

$$\begin{aligned}\mathbf{vol}_{G^i}(D^i_j) = \mathbf{vol}_{G^i[U^i_j]}(D^i_j) + w(E_{G^i}(D^i_j, U^i \setminus U^i_j)) &\overset{(6.7)}{\leq} \frac{1}{\phi} w(\partial_{G^i[U^i_j]} D^i_j) + w(E_{G^i}(D^i_j, U^i \setminus U^i_j)) \\ &\leq \frac{1}{\phi}\Big(w(\partial_{G^i[U^i_j]} D^i_j) + w(E_{G^i}(D^i_j, U^i \setminus U^i_j))\Big) \\ &= \frac{1}{\phi} w(\partial_{G^i} D^i_j),\end{aligned}$$

so summing over all $j \in [k_i]$,

$$\sum_{j\in[k_i]} \frac{1}{\phi} w(\partial_{G^i} D^i_j) \geq \sum_{j\in[k_i]} \mathbf{vol}_{G^i}(D^i_j) > \frac{\tau\lambda}{\phi}.$$

By Lemma 6.5.4, it follows that

$$w(\partial_G S) \geq \beta^{O(L)} \sum_{j\in[k_i]} w(\partial^i_G D^i_j) \geq \beta^{O(L)}\tau\lambda.$$

□

We now set some of our parameters; see Figure 6.1 for a complete table of the parameters in our proof. For $r := (\log n)^{1/6}$, let $\beta := (\log n)^{-O(r^4)}$ and $\phi := (\log n)^{-r^5}$, so that by Theorem 6.5.2, the total weight of inter-cluster edges, and therefore the total weight of the next graph in the expander decomposition sequence, shrinks by factor $(\log n)^{O(r^4)}\phi = (\log n)^{-\Omega(r^5)}$. Since edge weights are assumed to be polynomially bounded, this shrinking can only happen $O(\frac{\log n}{r^5})$ times, so $L \leq O(\frac{\log n}{r^5})$.

Let $\widetilde{\lambda} \in [\lambda, 3\lambda]$ be a 3-approximation to the mincut, computable in $\tilde{O}(m)$ time [80], Let $\epsilon' := \frac{1}{2}(\frac{\phi}{(L+1)\tau})^2\epsilon$ for parameter τ that we set later, and let $\widehat{H}$ be the sparsifier of τ-

Par.	Value
λ	Mincut of G
$\widetilde{\lambda}$	3-approximation of λ
ϵ	Given as input
r	$(\log n)^{1/6}$
β	$(\log n)^{-O(r^4)}$ from Theorem 6.5.2
ϕ	$(\log n)^{-r^5}$
L	$O(\frac{\log n}{r^5})$
γ	$2^{O(\log n)^{5/6}(\log\log n)^{O(1)}}$ from Theorem 6.5.14
Δ	$2^{\Theta(\log n)^{5/6}}$ from Theorem 6.5.14
τ	$\beta^{-cL}\gamma^2/\epsilon$ for large enough constant $c > 0$
ϵ'	$\frac{1}{2}(\frac{\phi}{(L+1)\tau})^2\epsilon$
$\widehat{W}$	$\min\{\frac{C\epsilon'\phi\widetilde{\lambda}}{\tau\ln(Lm)}, \frac{\widetilde{\lambda}}{\Delta}\}$ where $C > 0$ is the constant from Lemma 6.5.9
$\widetilde{W}$	$\frac{\epsilon}{2\gamma}\cdot\frac{\widetilde{\lambda}}{\Delta}$

Figure 6.1: The parameters in the proof of Theorem 6.5.1.

unbalanced cuts from Lemma 6.5.9 for this value of ϵ' (instead of ϵ) and the following value of $\widehat{W} \le \frac{C\epsilon'\phi\lambda}{\tau\ln(Lm)}$ (taking the place of W):

$$\widehat{W} := \min\left\{\frac{C\epsilon'\phi\widetilde{\lambda}}{3\tau\ln(Lm)}, \frac{\widetilde{\lambda}}{\Delta}\right\} = \min\left\{\Omega\left(\frac{\epsilon\phi^3\widetilde{\lambda}}{\tau^3L^2\ln(Lm)}\right), \frac{\widetilde{\lambda}}{\Delta}\right\}.$$

Let $\widetilde{H}$ be the unweighted graph from Theorem 6.5.14 applied to $\widetilde{\lambda}$ and Δ, so that $\widetilde{\lambda}/\Delta\cdot\widetilde{H}$ is a γ-approximate cut sparsifier for $\gamma := 2^{O(\log n)^{5/6}(\log\log n)^{O(1)}}$. Define $\widetilde{W} := \frac{\epsilon}{2\gamma}\cdot\frac{\widetilde{\lambda}}{\Delta}$, and let H' be the "union" of the graph $\widehat{H}$ weighted by $\widehat{W}$ and the graph $\widetilde{H}$ weighted by $\widetilde{W}$. More formally, consider a weighted graph H' where each edge (u, v) is weighted by $\widehat{W}\cdot w_{\widehat{H}}(u, v)+\widetilde{W}\cdot w_{\widetilde{H}}(u, v)$.

For an τ-unbalanced cut ∂S, the addition of the graph $\widetilde{H}$ weighted by $\widetilde{W}$ increases its weight by

$$\widetilde{W}\cdot w(\partial_{\widetilde{H}}S) = \frac{\epsilon}{2\gamma}\cdot\left(\frac{\lambda}{\Delta}w(\partial_{\widetilde{H}}S)\right) \le \frac{\epsilon}{2\gamma}\cdot\gamma w(\partial_G S) = \frac{\epsilon}{2}w(\partial_G S),$$

so that

$$\begin{aligned}\left|w(\partial_G S) - \left(\widehat{W}\cdot w(\partial_{\widehat{H}}S) + \widetilde{W}\cdot w(\partial_{\widetilde{H}}S)\right)\right| &\le \left|w(\partial_G S) - \widehat{W}\cdot w(\partial_{\widehat{H}}S)\right| + \widetilde{W}\cdot w(\partial_{\widetilde{H}}S^*)\\ &\le \left(\frac{(L+1)\tau}{\phi}\right)^2\cdot\epsilon'\lambda + \frac{\epsilon}{2}w(\partial_G S)\\ &= \frac{\epsilon\lambda}{2} + \frac{\epsilon}{2}w(\partial_G S)\\ &\le \epsilon w(\partial_G S).\end{aligned}$$

In particular, any τ-unbalanced cut satisfies

$$(1-\epsilon)\lambda \le \widehat{W} \cdot w(\partial_{\widehat{H}} S) + \widetilde{W} \cdot w(\partial_{\widetilde{H}} S) \le (1+\epsilon)\lambda. \tag{6.19}$$

Next, we show that all balanced cuts have weight at least λ in the graph $\widetilde{H}$ weighted by $\widetilde{W}$. This is where we finally set $\tau := \beta^{-cL}\gamma^2/\epsilon$ for large enough constant $c > 0$. For a balanced cut S,

$$\widetilde{W} \cdot w(\partial_{\widetilde{H}} S) = \frac{\epsilon}{2\gamma} \cdot \left(\frac{\lambda}{\Delta} w(\partial_{\widetilde{H}} S) \right) \ge \frac{\epsilon}{2\gamma} \cdot \left(\frac{1}{\gamma} w(\partial_G S) \right) \overset{\text{Clm.6.5.15}}{\ge} \frac{\epsilon}{\gamma^2} \cdot \beta^{O(L)} \tau\lambda \ge \lambda.$$

Moreover, by Claim 6.5.6 for this value of $\tau \ge \beta^{-O(L)}$, the mincut ∂S^* is τ-unbalanced, and therefore has weight at least $(1-\epsilon)\lambda$ in H' by (6.19).

Therefore, H' preserves the mincut up to factor ϵ and has mincut at least $(1-\epsilon)\lambda$. It remains to make all edge weights the same on this sparsifier. Since $\widetilde{W} = \frac{\epsilon}{2\gamma} \cdot \frac{\widetilde{\lambda}}{\Delta}$ and the only requirement for Δ from Theorem 6.5.14 is that $\Delta \ge 2^{O(\log n)^{5/6}}$, we can increase or decrease Δ by a constant factor until either $\widetilde{W}/\widehat{W}$ or $\widehat{W}/\widetilde{W}$ is an integer. Then, we can let $W := \min\{\widehat{W}, \widetilde{W}\}$ and define the unweighted graph H so that $\#_H(u,v) = w_{H'}(u,v)/W$ for all $u, v \in V$. Therefore, our final weight W is

$$\begin{aligned} W = \min\{\widehat{W}, \widetilde{W}\} &= \min\left\{ \Omega\left(\frac{\epsilon\phi^3\widetilde{\lambda}}{\tau^3 L^2 \ln(Lm)} \right), \frac{\widetilde{\lambda}}{\Delta}, \frac{\epsilon}{2\gamma} \cdot \frac{\widetilde{\lambda}}{\Delta} \right\} \\ &\ge \epsilon^4 2^{-O(\log n)^{5/6} (\log\log n)^{O(1)}} \lambda, \end{aligned}$$

so we can set $f(n) := 2^{O(\log n)^{5/6}(\log\log n)^{O(1)}}$, as desired.

Finally, we bound the running time. The expander decomposition sequence (Theorem 6.5.2) takes time $m^{1+O(1/r)} + \tilde{O}(m/\phi^2)$, the unbalanced case (Theorem 6.5.2) takes time $\tilde{O}(L^2 m)$, and the balanced case takes time $2^{O(\log n)^{5/6}(\log\log n)^{O(1)}} m$. Altogether, the total is $2^{O(\log n)^{5/6}(\log\log n)^{O(1)}} m$, which concludes the proof of Theorem 6.5.1.

6.5.5 Removing the Maximum Weight Assumption

Let $f(n) = 2^{O(\log n)^{5/6}(\log\log n)^{O(1)}}$ be the function from Theorem 6.5.1. In this section, we show how to use Theorem 6.5.1, which assumes that the maximum edge weight in G is at most $\epsilon^4\lambda/f(n)$, to prove Theorem 6.3.4, which makes no assumption on edge weights.

First, we show that we can assume without loss of generality that the maximum edge weight in G is at most 3λ. To see why, the algorithm can first compute a 3-approximation $\widetilde{\lambda} \in [\lambda, 3\lambda]$ to the mincut with the $\tilde{O}(m)$-time $(2+\epsilon)$-approximation algorithm of Matula [80], and for each edge in G with weight more than $\widetilde{\lambda}$, reduce its weight to $\widetilde{\lambda}$. Let the resulting graph be $\widetilde{G}$. We now claim the following:

Claim 6.5.16

Suppose an unweighted graph H and some weight W satisfy the two properties of Theorem 6.3.4 for $\widetilde{G}$. Then, they also satisfy the two properties of Theorem 6.3.4 for G.

Proof. The only cuts that change value between G and $\widetilde{G}$ are those with an edge of weight more than $\widetilde{\lambda}$, which means their value must be greater than $\widetilde{\lambda} \geq \lambda$. In particular, since G and $\widetilde{G}$ have the same mincuts and the same mincut values, both properties of Theorem 6.3.4 also hold when the input graph is G. $\square$

For the rest of the proof, we work with $\widetilde{G}$ instead of G. Define $\widetilde{W} := \epsilon^4\widetilde{\lambda}/(3f(n))$, which satisfies $\widetilde{W} \leq \epsilon^4\lambda/f(n)$. For each edge e in $\widetilde{G}$, split it into $\lceil w(e)/\widetilde{W}\rceil$ parallel edges of weight at most $\widetilde{W}$ each, whose sum of weights equals $w(e)$; let the resulting graph be $\widehat{G}$. Apply Theorem 6.5.1 on $\widehat{G}$, which returns an unweighted graph H and weight $W \geq \epsilon^4\lambda/f(n)$ such that the two properties of Theorem 6.3.4 hold for $\widehat{G}$. Clearly, the cuts are the same in $\widetilde{G}$ and $\widehat{G}$: we have $w(\partial_{\widetilde{G}}S) = w(\partial_{\widehat{G}}S)$ for all $S \subseteq V$. Therefore, the two properties also hold for $\widehat{G}$, as desired.

We now bound the size of G' and the running time. Since $w(e) \leq \widetilde{\lambda}$, we have $\lceil w(e)/\widetilde{W}\rceil \leq \lceil 3f(n)/\epsilon^4\rceil$, so each edge splits into at most $O(f(n)/\epsilon^4)$ edges and the total number of edges is $\widehat{m} \leq O(f(n)/\epsilon^4)\cdot m$. Therefore, Theorem 6.5.1 takes time $2^{O(\log n)^{5/6}(\log\log n)^{O(1)}}\widehat{m} = \epsilon^{-4}2^{O(\log n)^{5/6}(\log\log n)^{O(1)}}m$, concluding the proof of Theorem 6.3.4.

6.6 Conclusion

In this chapter, we presented a deterministic, almost-linear time algorithm for global mincut. One immediate open question is whether the running time can be improved to $m\,\mathrm{polylog}(n)$ to match the randomized complexity up to $\mathrm{polylog}(n)$ factors. This direction has two significant obstacles, however. The first, which is discussed in Section 8.9, is that the deterministic expander decomposition algorithm already takes $m^{1+o(1)}$ time, and improving even that to $m\,\mathrm{polylog}(n)$ would require significantly new ideas. Moreover, even if that were accomplished, the nature of our boundary-linked expander decomposition hierarchy would still incur an additional $2^{O(\sqrt{\log n\log\log n})}$ factor, so the overall running time is still $m2^{O(\sqrt{\log n\log\log n})}$. Bypassing the expander decomposition hierarchy would itself require novel ideas that may see applications to other graph cut problems.

Chapter 7

Parallel Shortest Path

In this chapter, we discuss our preconditioning-based approach to computing approximate shortest paths in parallel. Our main result is a parallel algorithm to compute $(1+\epsilon)$-approximate single-source shortest paths in $m\,\text{polylog}(n)$ work and $\text{polylog}(n)$ time, based on the work of [68].

We approach this problem from a continuous perspective by studying the closely related *minimum transshipment* problem, which we view as a continuous relaxation of the single-source shortest paths problem. To solve minimum transshipment, we combine preconditioning with *iterative methods* that minimize continuous functions in a small number of parallel rounds. Notably, our approach deviates from the previous *hopset*-based shortest path algorithms which come close to, but do not quite attain, the targeted $m\,\text{polylog}(n)$ work and $\text{polylog}(n)$ time. In other words, this chapter serves as evidence that preconditioning-based methods, when combined with the inherent parallelism of iterative methods, is a promising research direction in parallel graph algorithms.

Our algorithm for $(1+\epsilon)$-approximate minimum transshipment follows the preconditioning-based framework of Sherman [97]. In the context of graph distance algorithms, the well-conditioned graphs are precisely the low-diameter graphs, and Sherman's key insight is that there is a simple transshipment algorithm on such graphs. To generalize the algorithm to all instances, we compute low-diameter decompositions—the distance-based equivalent of expander decompositions—at varying diameter scales. To control the errors over all scales, we actually compute the low-diameter decompositions on an *embedding* of the graph into high-dimensional Euclidean space. This part of our algorithm improves upon Sherman's original transshipment algorithm and is required to achieve the desired $m\,\text{polylog}(n)$ work and $\text{polylog}(n)$ time. Finally, as Sherman's algorithm only works for the sequential setting, we face new challenges in developing a fully parallel algorithm. We end up adopting a recursive framework, reducing the minimum transshipment problem to sufficiently smaller instances of itself.

7.1 Background

The single-source shortest path problem is one of the most fundamental combinatorial optimization problems, and is also among the most notorious in parallel computation models. While the sequential model has simple near-linear time algorithm dating back to Dijkstra, much remains unknown for even the PRAM model despite decades of extensive research.

One of the most well-known settings studied so far in the PRAM model is the case of $(1+\epsilon)$-approximate single-source shortest paths in undirected graphs. Early work on this problem produced algorithms in sublinear time [61, 62], until the breakthrough result of Cohen [28], who presented an algorithm in $O(m^{1+\epsilon_0})$ work (for any constant $\epsilon_0 > 0$) and $\text{polylog}(n)$ time through the use of *hopsets*: additional edges added to the graph so that short paths in the graph span few edges. Since then, it was a long-standing open problem whether Cohen's algorithm could be improved to run in $m\,\text{polylog}(n)$ work while keeping the time $\text{polylog}(n)$.

Recently, this question was partially answered by Abboud, Bodwin and Pettie [1], surprisingly in the negative: they showed that there exist families of graphs for which any hopsets on these graphs must have size $\Omega(m^{1+\epsilon_0})$, thereby lower bounding the work by $\Omega(m^{1+\epsilon_0})$ for any purely hopset-based algorithm like Cohen's. While their lower bound does not rule out other approaches to this problem, no other directions of attack have come close to matching Cohen's method of hopsets before the results of this chapter and the concurrent work of Andoni, Stein, and Zhong [9] on the same problem.

7.1.1 Our Contributions

In this chapter, we tackle this problem from a new perspective: *continuous optimization*, especially the methods pioneered by Sherman [96] for the max-flow problem. By reducing to studying the closely-related and more continuous *minimum transshipment* problem, we provide the first $(1+\epsilon)$-approximate SSSP algorithm for weighted, undirected graphs in $m\,\text{polylog}(n)$ work and $\text{polylog}(n)$ time in the PRAM model, bypassing the hopset lower bound and resolving the aforementioned open problem. This serves as evidence that continuous optimization, with its rich theory in graph algorithm and inherent parallelism, is a promising research direction in parallel graph algorithms and can bypass known barriers to other common approaches.

Theorem 7.1.1: Parallel SSSP

There exists a parallel algorithm that, given an undirected graph with nonnegative weights, computes a $(1+\epsilon)$-approximate single-source shortest path tree in $m\,\text{polylog}(n)\,\epsilon^{-2}$ work and $\text{polylog}(n)\,\epsilon^{-2}$ time in the PRAM model.

Our SSSP algorithm is recursive, cycling through three problems in a round-robin fashion: SSSP, transshipment, and the problem of computing an ℓ_1-embedding of a graph with polylog(n) distortion in $O(\log n)$ dimensions. That is, each problem calls the next problem on the cyclic list possibly many times, and possibly on a smaller graph instance. Hence, we obtain parallel algorithms with similar running times for the other two problems as well.

Theorem 7.1.2: Parallel transshipment

There exists a parallel algorithm that, given an undirected graph with nonnegative weights and polynomial aspect ratio, computes a $(1+\epsilon)$-approximation to minimum transshipment in $m\,\text{polylog}(n)\,\epsilon^{-2}$ work and $\text{polylog}(n)\,\epsilon^{-2}$ time in the PRAM model.

Theorem 7.1.3: Parallel ℓ_1-embedding

There exists a parallel algorithm that, given an undirected graph with nonnegative weights and polynomial aspect ratio, computes an ℓ_1-embedding with polylog(n) distortion in $O(\log n)$ dimensions in $m\,\text{polylog}(n)$ work and polylog(n) time in the PRAM model.

Theorem 7.1.2 also establishes the first $m\,\text{polylog}(n)$ time *sequential* algorithm for $(1+\epsilon)$-approximate transshipment, improving upon the $m^{1+o(1)}$-time algorithm of Sherman [97]. For readers primarily interested in the sequential setting, we further optimize our parameters to the following. Note that the best algorithm for the closely-related max-flow problem [90] requires $O(m\log^{41} n)$ time in comparison. Our algorithm is also technically considerably simpler than the max-flow algorithm, and may serve as a gentler introduction to readers new to continuous optimization methods in graph algorithms.

Theorem 7.1.4: Sequential transshipment

There is an algorithm that, given an undirected graph with nonnegative weights and polynomial aspect ratio, computes a $(1+\epsilon)$-approximation to minimum transshipment in time $O((m\log^{10} n + n\log^{15} n)\,\epsilon^{-2}\,(\log\log n)^{O(1)})$.

7.1.2 Our Techniques

We follow Sherman's preconditioning-based approach for transshipment [97]. Sherman's framework reduces the problem of approximate transshipment to that of ℓ_1-*oblivious routing* in a matrix-theoretic sense. More precisely, the task is to compute a sparse matrix R such that for any transshipment demand vector b, the value $\|Rb\|_1$ approximates up to polylogarithmic factors the minimum transshipment cost with demands b. Note that if the graph has aspect ratio Δ, then setting R as the square identity matrix approximates the minimum transshipment cost up to factor 2Δ. (To see this, first consider the case when $b = \mathbb{1}_s - \mathbb{1}_t$, and then decompose a general demand into such vectors according to the minimum transshipment flow.) In other words, for average-case instances with low aspect ratio, computing a good matrix R is trivial. To handle general graphs, we first compute an ℓ_1-embedding and essentially phrase the question purely from a geometric point of view, which was also Sherman's approach [97]. We then decompose the embedded vertices into grid cells, which can be viewed as a graph distance analogue of expander decomposition. Our key new insight is to randomly *shift* the grid when building the grid cells, a technique borrowed from low-dimensional computational geometry [49]. The ℓ_1-oblivious routing algorithm is mostly self-contained and has no relation to the parallel sections of the chapter. We therefore isolate it in its own section, Section 7.4, for the convenience of readers primarily interested in transshipment in the sequential setting.

Our recursive algorithm is inspired by a similar recursive algorithm by Peng [90] for max-flow. It is instructive to compare our result to that of Peng [90], the first $\tilde{O}(m)$ time[1] algorithm for $(1-\epsilon)$-approximate max-flow.[2] Peng [90] uses an oblivious routing scheme for max-flow that achieves polylog(n)-approximation, but requires polylog(n) calls to $(1-\epsilon)$-max-flow [91]. This oblivious routing scheme produced a chicken-and-egg situation for max-flow and oblivious routing, since each one required calls to the other. Peng's main contribution is breaking this cycle, by allowing the oblivious routing to call max-flow on sufficiently smaller-sized graphs to produce an efficient recursive algorithm. Here, we adopt a similar recursive approach, cycling through the problems of shortest path, minimum transshipment, oblivious routing, and ℓ_1-embedding.

Step 1: reduce to transshipment. The first step of the algorithm is to reduce the approximate SSSP problem to the approximate *minimum transshipment* problem, which was previously done in [14] for various other computational models. Making it work in the PRAM model requires a little more care, and for completeness, we provide a self-contained reduction in Sections 7.8 and 7.9.

Note that if we were in the exact case, then the reduction would be immediate: there is a straightforward reduction from exact SSSP to exact transshipment: set $-(n-1)$ demand on the source vertex and $+1$ demand on the rest, and from the transshipment flow we can recover the exact SSSP relatively easily. However, in the approximate case, an approximate transshipment solution in the same reduction only satisfies distances on "average". [14] handles this issue through $O(\log n)$ calls to approximate transshipment with carefully and adaptively constructed demands on each call; we use $O(\log^2 n)$ calls instead with a more sophisticated reduction.

Step 2: ℓ_1-oblivious routing. As mentioned before, Sherman's framework reduces the problem of approximate transshipment to that of ℓ_1-*oblivious routing*. We follow the same

[1]Throughout this chapter, we use the standard $\tilde{O}(\cdot)$ notation to hide polylogarithmic factors in the running time.

[2]Note that max-flow and minimum transshipment are closely related: for graph incidence matrix A and a diagonal matrix C capturing the edge capacities/costs, and for a given demand vector b, the max-flow problem is equivalent to $\min \|f\|_\infty$ subject to $Af = b$, and the minimum transshipment problem is exactly $\min \mathbb{1}f$ subject to $Af = b$.

approach in Section 7.4, computing an ℓ_1-embedding matrix R given an initial ℓ_1-embedding of the graph with polylog(n) distortion.

Step 3: ℓ_1-embedding and ultra-sparsification. Unfortunately, while ℓ_1-embeddings are simple to compute sequentially, no work-efficient parallel algorithm is known. This is because the popular algorithms that compute ℓ_1-embeddings sequentially all require distance computations as subroutines, and no work-efficient parallel algorithm for SSSP is known. (If one were known, then there would be no need for our result in the first place!)

Recall that we sought out to solve SSSP, and currently, our ℓ_1-embedding problem *requires* an SSSP routine on its own. This is where Peng's key insight comes to play: while recursing naively on the same graph will not work (since it would loop endlessly), if we can recurse on *sufficiently smaller* graphs, then the recursion analysis would produce an algorithm with the desired running time. This is indeed Peng's approach for max-flow: he makes one max-flow instance call ℓ_∞-oblivious routing, which in turn calls max-flow a number of times, but ensures that the total size of the recursive calls is at most half the size of the original graph. The recursion then works out to roughly $T(m) = \sum_i T(m_i) + \tilde{O}(m)$ where $\sum_i m_i \le m/2$, which solves to $T(m) = \tilde{O}(m)$.

How does Peng achieve the reduction in size? Instead of computing ℓ_∞-oblivious routing in the original graph G, he first *(edge-)sparsifies* G into a graph H on n vertices and $(n-1) + O(\frac{m}{\text{polylog}(n)})$ edges by computing an *ultra-sparsifier* of the graph [63]. This is a graph that is so sparse that it is almost "tree-like" (at least when $m = \tilde{O}(n)$). Of course, this alone might not achieve the desired size reduction, for example if $m \approx n$. Therefore, he next *vertex-sparsifies* H into a graph H' with $O(\frac{m}{\text{polylog}(n)})$ vertices and $O(\frac{m}{\text{polylog}(n)})$ edges using a j-tree construction of Madry [77]. He now calls ℓ_∞-oblivious routing on H' (instead of G), which again calls max-flow, but this time on graphs of small enough size (w.r.t. the original graph) to make the recursion work out. Moreover, by the properties of the ultra-sparsifier and the vertex-sparsifier, a polylog(n)-approximate ℓ_∞-oblivious routing scheme for H' is also a polylog(n)-approximate ℓ_∞-oblivious routing scheme for G (that is, the approximation suffers an extra polylog(n) factor). The specific polylog(n) factor does not matter at the end, since in Sherman's framework, any polylog(n) factor is sufficient to boost the error to $(1+\epsilon)$ for max-flow at an additional *additive* cost of $\tilde{O}(m)$.

Our approach is similar, but adapted from ℓ_∞/max-flow to ℓ_1/transshipment. The ℓ_1-analogy of an ultrasparsifier has been studied previously by Elkin and Neiman [34], who coined the term *ultra-sparse spanner*; in this chapter, we will use *ultra-spanner* instead to emphasize its connection to ultra-sparsifiers. Instead of running ℓ_1-embedding on G, we compute an ultra-spanner H, and then vertex-sparsify it in the same manner as Peng; again, the resulting graph H' has $O(\frac{m}{\text{polylog}(n)})$ vertices and edges. We then run ℓ_1-embedding on H', making calls to (approximate) SSSP on graphs of much smaller size. It turns out that approximate SSSP works for the ℓ_1-embedding algorithm that we use, provided that the distances satisfy a certain triangle inequality condition that our SSSP algorithm obtains for free.

7.1.3 Chapter Organization

In Section 7.3, we introduce the high-level components of our recursive parallel algorithm (see Figure 7.1), leaving the details to later sections and the appendix.

Section 7.4 is focused exclusively on the sequential transshipment result (Theorem 7.1.4). The algorithm is almost completely self-contained, save for Sherman's framework and an initial ℓ_1-embedding step (which can be computed quickly sequentially [74]). It has nothing deferred to the appendix in an attempt to make it a standalone section for readers primarily interested in Theorem 7.1.4.

7.2 Additional Preliminaries

All graphs in this chapter are undirected and (positively) weighted, with the exception of Section 7.8, where directed graphs and edges of zero weights are defined explicitly. For two vertices $u, v \in V(G)$, we define $d_G(u, v)$ as the (weighted) distance between u and v in G; if the graph G is clear from context, we sometimes use $d(u, v)$ instead.

7.2.1 PRAM Model

Our PRAM model is based off of the one in [37], also called the *work-span* model. An algorithm in the PRAM model proceeds identically to a sequential algorithm except for the addition of the parallel foreach loop. In a parallel foreach, each iteration of the loop must run independently of the other tasks, and the parallel algorithm may execute all iterations in parallel instead of sequentially. The *work* of a PRAM algorithm is the same as the sequential running time if each parallel foreach was executed sequentially instead. To determine the *time* of the algorithm, for every parallel foreach, we calculate the maximum sequential running time over all iterations of the loop, and sum this quantity over all parallel foreach loops. We then add onto the total the sequential running time outside the parallel foreach loops to determine the total time. There are different variants of the PRAM model, such as the binary-forking model and the unlimited forking model, that may introduce additional overhead in foreach loops. However, these all differ by at most polylogarithmic factors in their work and span, which we always hide behind $\tilde{O}(\cdot)$ notation, so we do not concern ourselves with the specific model.

7.2.2 Transshipment Preliminaries

The definitions below are central for our sequential transshipment algorithm (Theorem 7.1.4, Section 7.4) and are also relevant for the parallel algorithms.

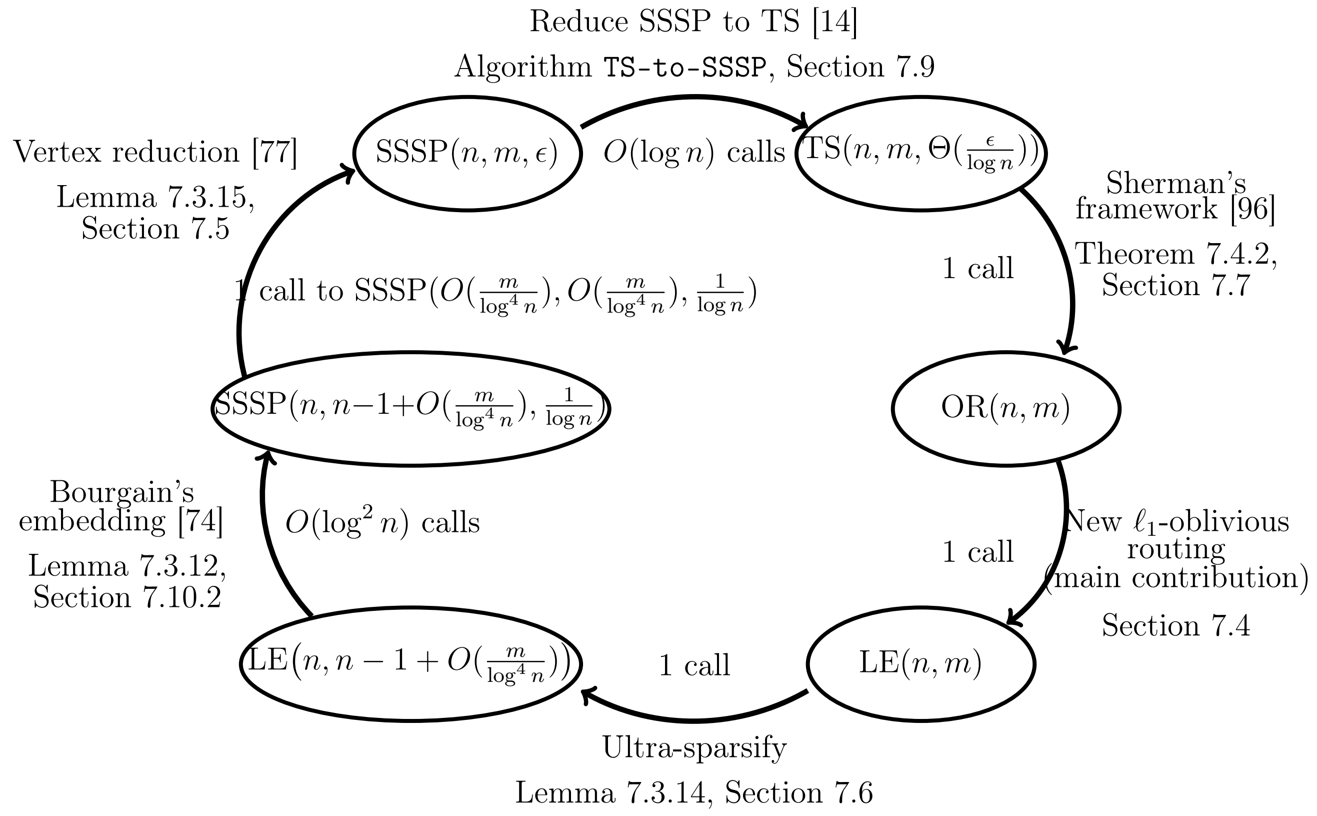
Reduce SSSP to TS [14]
Algorithm TS-to-SSSP, Section 7.9
$\mathrm{SSSP}(n, m, \epsilon)$
$O(\log n)$ calls
$\mathrm{TS}(n, m, \Theta(\frac{\epsilon}{\log n}))$
Sherman's framework [96]
Theorem 7.4.2, Section 7.7
1 call
$\mathrm{OR}(n, m)$
New ℓ_1-oblivious routing (main contribution)
Section 7.4
1 call
$\mathrm{LE}(n, m)$
1 call
Ultra-sparsify
Lemma 7.3.14, Section 7.6
$\mathrm{LE}(n, n-1+O(\frac{m}{\log^4 n}))$
Bourgain's embedding [74]
Lemma 7.3.12, Section 7.10.2
$O(\log^2 n)$ calls
$\mathrm{SSSP}(n, n-1+O(\frac{m}{\log^4 n}), \frac{1}{\log n})$
1 call to $\mathrm{SSSP}(O(\frac{m}{\log^4 n}), O(\frac{m}{\log^4 n}), \frac{1}{\log n})$
Vertex reduction [77]
Lemma 7.3.15, Section 7.5

Figure 7.1: Our recursive approach, inspired by [90]'s for max-flow. $\mathrm{SSSP}(n, m, \epsilon)$ is the work required to compute $(1+\epsilon)$-approximate SSSP (on a graph with n vertices and m edges) that satisfies a certain triangle inequality condition that we omit here. $\mathrm{TS}(n, m, \epsilon)$ is the work required to compute $(1+\epsilon)$-approximate transshipment. $\mathrm{OR}(n, m)$ is the work required to compute a polylog(n)-approximate ℓ_1-oblivious routing (matrix), and $\mathrm{LE}(n, m)$ is the work required to compute an ℓ_1-embedding in $O(\log n)$ dimensions with at most polylog(n) distortion.

Definition 7.2.1: Transshipment

The *minimum transshipment* problem inputs a (positively) weighted, undirected graph $G = (V, E)$, and defines the following auxiliary matrices:

1. Incidence matrix $A \in \mathbb{R}^{V \times E}$: for each edge $e = (u, v)$, the column of A indexed by e equals either $\mathbb{1}_u - \mathbb{1}_v$ or $\mathbb{1}_v - \mathbb{1}_u$.
2. Cost matrix $C \in \mathbb{R}^{E \times E}$: a diagonal matrix with entry $C_{e,e}$ equal to the weight of edge e.

In a *transshipment instance*, we are also given a *demand vector* $b \in \mathbb{R}^V$ satisfying $\mathbb{1}^T b = 0$.

Consider now the LP formulation for minimum transshipment: $\min \mathbb{1}Cf : Af = b$, and its dual, $\max b^T\phi : \left\|C^{-1}A^T\phi\right\|_\infty \le 1$. Let us define the solutions to the primal and dual formulations as *flows* and *potentials*:

Definition 7.2.2: Flow

Given a transshipment instance, a *flow vector* (or *flow*) is a vector $f \in \mathbb{R}^E$ satisfying the *primal constraints* $Af = b$, and it has *cost* $\mathbb{1}Cf$. The flow minimizing $\mathbb{1}Cf$ is called the *optimal flow* of the transshipment instance. For any $\alpha \ge 1$, an *α-approximate flow* is a flow whose value $\mathbb{1}Cf$ is at most α times the minimum possible (over all flows).

Definition 7.2.3: Potential

Given a transshipment instance, a *set of potentials* (or *potential*) is a vector $\phi \in \mathbb{R}^V$ satisfying the *dual constraints* $\left\|C^{-1}A^T\phi\right\|_\infty \le 1$. The potential maximizing $b^T\phi$ is called the *optimal potential* of the transshipment instance.

For convenience, we will treat potentials as functions on V; that is, we will use the notation $\phi(v)$ instead of ϕ_v.

Definition 7.2.4: Flow-potential pair

For any flow $f \in \mathbb{R}^E$ and potential $\phi \in \mathbb{R}^V$, the pair (f, ϕ) is called a *flow-potential pair*. For $\alpha \ge 1$, (f, ϕ) is an *α-approximate flow-potential pair* if $\mathbb{1}Cf \le \alpha\, b^T\phi$.

Fact 7.2.5

If (f, ϕ) is an α-approximate flow-potential pair, then f is an α-approximate flow.

Proof. Let f^* be the optimal flow. The two LPs $\min \mathbb{1}Cf : Af = b$ and $\max b^T\phi : \left\|C^{-1}A^T\phi\right\|_\infty \le 1$ are duals of each other, so by (weak) LP duality, the potential ϕ satisfies $b^T\phi \le \mathbb{1}Cf^*$. Since (f, ϕ) is an α-approximate flow-potential pair, we have $\mathbb{1}Cf \le \alpha\, b^T\phi \le \alpha\mathbb{1}Cf^*$. □

Definition 7.2.6: opt

Given a transshipment problem and demand vector b, define $\mathsf{opt}(b)$ as the cost of the optimal flow of that instance, that is:

$$\mathsf{opt}(b) := \min_{f:Af=b} \mathbb{1}Cf.$$

When the underlying graph G is ambiguous, we use the notation $\mathsf{opt}_G(b)$ instead.

7.2.3 Parallel Shortest Path Preliminaries

The definitions below are confined to the parallel algorithms in this chapter, so a reader primarily interested in the sequential transshipment algorithm (Theorem 7.1.4, Section 7.4) may skip these.

We first introduce a notion of approximate SSSP distances which we call *approximate SSSP potentials*.

Definition 7.2.7: Approximate s-SSSP potential

Given a graph $G = (V, E)$ and a source s, a vector $\phi \in \mathbb{R}^V$ is an **α-approximate s-SSSP potential** if:

1. For all $v \in V$, $\phi(v) - \phi(s) \geq \frac{1}{\alpha} \cdot d(s, v)$
2. For each edge (u, v), $|\phi(u) - \phi(v)| \leq w(u, v)$.

When the source s is either irrelevant or clear from context, we may use *α-approximate SSSP potential* (without the s) instead.

Observe that the approximate SSSP potential problem is slightly more stringent than simply approximate shortest path distances: the second condition of Definition 7.2.7 requires that distances satisfy a sort of approximate *subtractive* triangle inequality. To illustrate why this condition is more restrictive, imagine a graph on three vertices s, u, v, with $d(s, u) = d(s, v) = 100$ and $d(u, v) = 1$, and let $\alpha := 10/9$. Then, the distance estimates $\tilde{d}(s) = 0$ and $\tilde{d}(u) = 90$ and $\tilde{d}(v) = 100$ are α-approximate SSSP distances with source s, but the vector ϕ with $\phi(s) = 0$ and $\phi(u) = 90$ and $\phi(v) = 100$ is not a $(1 + \epsilon)$-approximate SSSP potential because it violates the second condition of Definition 7.2.7 for edge (u, v): we have $|\phi(u) - \phi(v)| = 10 > w(u, v) = 1$.

Observation 7.2.8

An α-approximate s-SSSP potential is also an α-approximate potential for the transshipment instance with demands $\sum_v (\mathbb{1}_v - \mathbb{1}_s)$ (but the converse is not true).

Observation 7.2.9

Given a graph $G = (V, E)$ and a source s, any α-approximate s-SSSP potential ϕ satisfies $|\phi(u) - \phi(v)| \leq d(u, v)$ for all $u, v \in V$.

Proof. Let $u = v_0, v_1, \ldots, v_\ell = v$ be the shortest path from s to v. By property (2), we have

$$|\phi(u) - \phi(v)| \leq \left| \sum_{i=1}^{\ell} \phi(v_i) - \phi(v_{i-1}) \right| \leq \sum_{i=1}^{\ell} |\phi(v_i) - \phi(v_{i-1})| \leq \sum_{i=1}^{\ell} d(v_i, v_{i-1}) = d(u, v).$$

□

Observation 7.2.10

If ϕ is an α-approximate s-SSSP potential, then $\phi + c \cdot \mathbb{1}$ is also one for any scalar $c \in \mathbb{R}$. Therefore, we can always assume w.l.o.g. that $\phi(s) = 0$. In that case, by property (1), we also have $\phi(v) \geq 0$ for all $v \in V$.

Observation 7.2.11

Given two vectors ϕ_1 and ϕ_2 that satisfy property (2), the vectors $\phi_{\min}$, $\phi_{\max} \in \mathbb{R}^V$ defined as $\phi_{\min}(v) := \min\{\phi_1(v), \phi_2(v)\}$ and $\phi_{\max}(v) := \max\{\phi_1(v), \phi_2(v)\}$ for all $v \in V$ also satisfy property (2).

We now generalize the notion of SSSP potential to the case when the "source" is a subset $S \subseteq V$, not a single vertex. Essentially, the definition is equivalent to contracting all vertices in S into a single source s, taking an s-SSSP potential, and setting the potential of each vertex in S to the potential of s.

Definition 7.2.12: Approximate S-SSSP potential

Given a graph $G = (V, E)$ and a vertex subset $S \subseteq V$, a vector $\phi \in \mathbb{R}^V$ is an **α-approximate S-SSSP potential** if:

0. For all $s \in S$, $\phi(s)$ takes the same value
1. For all $v \in V$ and $s \in S$, $\phi(v) - \phi(s) \geq \frac{1}{\alpha} \cdot d(s, v)$
2. For each edge (u, v), $|\phi(u) - \phi(v)| \leq w(u, v)$.

When the set S is either irrelevant or clear from context, we may use *α-approximate SSSP potential* (without the S) instead.

Observation 7.2.13

Given a graph $G = (V, E)$ and a vertex subset $S \subseteq V$, let G' be the graph with all vertices in S contracted into a single vertex s'. Then, if ϕ is an α-approximate S-SSSP potential, then the vector ϕ' defined as $\phi'(v) = v$ for $v \in V \setminus S$ and $\phi'(s') = \phi(s)$ for some $s \in S$ is an α-approximate s-SSSP potential in G'.

Also, we will need the notion of a *spanner* throughout this chapter:

Definition 7.2.14: Spanner

Given a graph $G = (V, E)$ and a parameter $\alpha \geq 1$, a subgraph $H \subseteq G$ is an *α-spanner* of G if for all $u, v \in V$, we have $d_G(u, v) \leq d_H(u, v) \leq \alpha\, d_G(u, v)$.

Polynomial Aspect Ratio Throughout this chapter, we assume that the initial input graph for the approximate SSSP problem has *polynomially bounded aspect ratio*, defined below:

Definition 7.2.15: Aspect ratio

The *aspect ratio* of a graph $G = (V, E)$ is the quantity $\frac{\max_{u,v \in V} d_G(u,v)}{\min_{u,v \in V} d_G(u,v)}$.

This assumption can be safely assumed: there is a reduction by Klein and Subramanian [61] (also used by Cohen [28]) that transforms the $(1+\epsilon)$-approximate SSSP problem on a graph with arbitrary, nonnegative weights to solving $(1+\epsilon/2)$-approximate SSSP on a collection of graphs of total size $O(m \log n)$, each with polynomially bounded aspect ratio, and requiring an additional $O(m \log n)$ work and $O(\log n)$ time. Since polynomially bounded aspect ratio is a common assumption in graph optimization problems, we will not present this reduction for sake of self-containment.

Since our SSSP algorithm is recursive, and the SSSP problem that we solve is actually the (slightly more general) SSSP potential problem, we do not apply the reduction of Klein and Subramanian again in each recursive call. Rather, we take some care to show that the aspect ratio does not blow up over recursive calls.

For the ℓ_1-embedding and transshipment problems, we will handle the aspect ratio issue differently. For the ℓ_1-embedding problem, we will explicitly require that the input graph has aspect ratio at most n^C for some fixed constant C (which can be made arbitrarily large). In particular, this assumption translates over in our theorem statement for parallel ℓ_1-embedding (Theorem 7.1.3). For the transshipment problem, we will not assume that the graph has polynomial aspect ratio, but every time we recursively call transshipment, we will ensure that the *demand vector* has small, integral entries in the recursive instance. Assuming this guarantee on the demand vector, we reduce the transshipment problem to the case when the graph also has polynomial aspect ratio like in the SSSP case, but here, the reduction is simple enough that we include it in this chapter for completeness (Lemma 7.3.5).

7.3 The Recursive Algorithm

Our algorithm will *recursively* cycle through three problems: approximate SSSP potentials, approximate transshipment, and ℓ_1-embedding. For the ℓ_1-embedding and SSSP potential problems, we will always assume that the input graph has aspect ratio at most n^C for some arbitrarily large but fixed constant $C > 0$ (that remains unchanged throughout the recursion). The transshipment problem will require no bound on aspect ratio: we provide a simple transformation on the graph to ensure that the aspect ratio is polynomial. Let us now define the work required to solve the three problems below:

1. $W_{\text{embed}}(m)$ and $T_{\text{embed}}(m)$ are the work and time to ℓ_1-embed a connected graph with m edges and aspect ratio at most n^5 into $O(\log n)$ dimensions with distortion $O(\log^{10.5} n)$, where the $O(\cdot)$ hides an arbitrarily large but fixed constant.
2. $W_{\text{SSSP}}(m,\epsilon)$ and $T_{\text{SSSP}}(m,\epsilon)$ are the work and time to compute an $(1+\epsilon)$-approximate SSSP potential of a connected graph with m edges and aspect ratio at most $\tilde{O}(n^5)$, where the $\tilde{O}(\cdot)$ hides a factor $c\log^c n$ for an arbitrarily large but fixed constant $c > 0$.
3. $W_{\text{TS}}(m,\epsilon)$ and $T_{\text{TS}}(m,\epsilon)$ are the work and time to compute a $(1+\epsilon)$-approximate transshipment instance of a connected graph with m edges, where the demand vector b is integral and satisfies $|b_v| \le n-1$ for all vertices v.

The following is the main result of Section 7.3.2:

Theorem 7.3.1: ℓ_1-embedding given SSSP on smaller instances

Let $G = (V,E)$ be a connected graph with n vertices and m edges with aspect ratio M, let $\beta \ge 1$ be a parameter, and let $\mathcal{A}$ be an algorithm that inputs (i) a connected graph on at most m/β vertices and edges with aspect ratio $\tilde{O}(\beta^2 M)$ and (ii) a source vertex s, and outputs a $(1+1/\log n)$-approximate s-SSSP potential. Then, there is an algorithm that computes an ℓ_1-embedding of G into $O(\log n)$ dimensions with distortion $O(\beta^2\log^{6.5} n)$ and calls $\mathcal{A}$ at most $O(\log^2 n)$ times in parallel, plus $\tilde{O}(m)$ additional work and $\operatorname{polylog}(n)$ additional time.

Corollary 7.3.2

$W_{\text{embed}}(m) \le O(\log^2 n)\cdot W_{\text{SSSP}}(\delta m/\log^4 n, 1/\log n) + \tilde{O}(m)$ for any fixed, arbitrarily small constant $\delta > 0$, and $T_{\text{embed}}(m) \le T_{\text{SSSP}}(\delta m/\log^4 n, 1/\log n) + \operatorname{polylog}(n)$.

Proof. Apply Theorem 7.3.1 with $\beta := \frac{1}{\delta}\log^2 n$, obtaining distortion $O(\beta^2\log^{6.5} n) = O(\log^{10.5} n)$. □

The following is a corollary of our sequential transshipment result in Section 7.4 which constitutes our main technical contribution of this chapter:

Corollary 7.3.3: Parallel SSSP given ℓ_1-embedding

Given an undirected graph with nonnegative weights and polynomial aspect ratio, and given an ℓ_1-embedding of the graph with polylog(n) distortion in $O(\log n)$ dimensions, there is a parallel algorithm to compute a $(1+\epsilon)$-approximate minimum transshipment instance in $\tilde{O}(m\epsilon^{-2})$ work and polylog$(n)\epsilon^{-2}$ time.

The following is Sherman's framework for the minimum transshipment problem, for which we provide a self-contained treatment through the *multiplicative weights* method in Section 7.7. This is where the error boosting takes place: given a lossy polylog(n)-approximate ℓ_1-oblivious routing algorithm encoded by the matrix R, we can boost the error all the way to $(1+\epsilon)$ for transshipment. The only overhead in Sherman's framework is an *additive* $\tilde{O}(m)$ work and polylog(n) time (where these polylogarithmic factors depend on the approximation of the ℓ_1-oblivious routing), which is ultimately what makes the recursion work out.

Theorem 7.3.4: Parallel Theorem 7.4.2 with extra $\log(n/\epsilon)$ factor

Given a transshipment problem, suppose we have already computed a matrix R satisfying:

1. For all demand vectors $b \in \mathbb{R}^n$,
$$\mathsf{opt}(b) \leq \mathbb{1}Rb \leq \kappa \cdot \mathsf{opt}(b) \tag{7.1}$$
2. Matrix-vector products with R and R^T can be computed in M work and polylog(n) time[a]

Then, for any transshipment instance with demand vector b, we can compute a flow vector f and a vector of potentials $\tilde{\phi}$ in $\tilde{O}(\kappa^2(m+n+M)\,\epsilon^{-2})$ time that satisfies:

1. $\left\|Cf\right\|_1 \leq (1+\epsilon)b^T\tilde{\phi} \leq (1+\epsilon)\,\mathsf{opt}(b)$
2. $\mathsf{opt}(Af-b) \leq \beta\,\mathsf{opt}(b)$

[a] M can potentially be much lower than the number of nonzero entries in the matrix R if it can be efficiently compressed.

Lastly, there is one minor mismatch: Corollary 7.3.3 assumes that the graph has polynomial aspect ratio, while the problem for $W_{\mathrm{TS}}(\cdot)$ does not assume such a thing, but rather assumes that the demand vector has entries restricted to $\{-(n-1), -(n-2), \ldots, n-2, n-1\}$. It turns out that given this restriction on the demand vector, the polynomial aspect ratio of the graph can be obtained for free. We defer this proof to Section 7.10.1.

Lemma 7.3.5: Aspect ratio guarantee

Given a transshipment instance with graph $G = (V, E)$ with n vertices and m edges and an integer demand vector b satisfying $|b_v| \leq M$ for all $v \in V$, we can transform G into another graph $\widehat{G}$ on n vertices and at most m edges such that $\widehat{G}$ has aspect ratio at most $n^4 M$, and $\mathsf{opt}_G(b) \leq \mathsf{opt}_{\widehat{G}}(b) \leq (1 + 1/n^2)\,\mathsf{opt}_G(b)$. The transformation takes $\tilde{O}(m)$ work and polylog(n) time.

Corollary 7.3.6

$W_{\text{TS}}(m, \epsilon) \leq W_{\text{embed}}(m) + \tilde{O}(m/\epsilon^2)$. That is, outside of an ℓ_1-embedding into $O(\log n)$ dimensions with distortion $O(\log^{10.5} n)$, the additional work to compute $(1 + \epsilon)$-approximate transshipment is $\tilde{O}(m/\epsilon^2)$, and the additional time is $\tilde{O}(1/\epsilon^2)$.

Proof. By assumption, the demand vector b_v is integral and satisfies $|b_v| \leq n - 1$ for all vertices v. Apply Lemma 7.3.5 with $M := n-1$ so that the aspect ratio of the modified graph $\widehat{G}$ is at most n^5, which is polynomial, and the optimal solution changes by factor at most $(1+1/n^2)$. Compute an ℓ_1-embedding of $\widehat{G}$ into polylog(n) dimensions (in $W_{\text{embed}}(m)$ work), and then apply Corollary 7.3.3 with approximation factor $(1+\epsilon/2)$. The final approximation factor is $(1 + 1/n^2)(1 + \epsilon/2)$, which is at most $(1 + \epsilon)$ for $\epsilon \geq \Omega(1/n^2)$. (If $\epsilon = O(1/n^2)$, then an algorithm running in time $\tilde{O}(1/\epsilon^2) \geq \tilde{O}(n^4)$ is trivial.) $\square$

We now present the reduction from approximate SSSP to approximate transshipment, partially inspired by a similar routine in [14]; for completeness, we give a self-contained proof of the reduction in Sections 7.8 and 7.9 in the form of this theorem:

Theorem 7.3.7

Let $G = (V, E)$ be a graph with n vertices and m edges, and let $\epsilon > 0$ be a parameter. Given graph G, a source $s \in V$, and an ℓ_1-embedding of it into $O(\log n)$ dimensions with distortion polylog(n), we can compute a $(1 + \epsilon)$-approximate SSSP tree and potential in additional $\tilde{O}(m/\epsilon^2)$ work and $\tilde{O}(1/\epsilon^2)$ time.

Corollary 7.3.8

$W_{\text{SSSP}}(m, \epsilon) \leq W_{\text{embed}}(m) + \tilde{O}(m/\epsilon^2)$ and $T_{\text{SSSP}}(m, \epsilon) \leq T_{\text{embed}}(m) + \tilde{O}(1/\epsilon^2)$.

Proof. This is essentially Theorem 7.3.7 in recursive form. $\square$

Corollary 7.3.9

$W_{\text{SSSP}}(m, \epsilon) \leq O(\log^2 n) \cdot W_{\text{SSSP}}(\delta m/\log^2 n, 1/\log n) + \tilde{O}(m/\epsilon^2)$ and $T_{\text{SSSP}}(m, \epsilon) \leq T_{\text{SSSP}}(\delta m/\log^2 n, 1/\log n) + \tilde{O}(1/\epsilon^2)$ for any fixed, arbitrarily small constant $\delta > 0$.

Proof. Follows directly from Corollaries 7.3.2 and 7.3.8. □

Corollary 7.3.10

$W_{\text{SSSP}}(m,\epsilon) \le \tilde{O}(m/\epsilon^2)$ and $T_{\text{SSSP}}(m,\epsilon) \le \tilde{O}(1/\epsilon^2)$.

Proof. Observe that in the recursion of Corollary 7.3.9, by setting $\delta > 0$ small enough, the total graph size $O(\log^2 n) \cdot \delta m / \log^2 n \le m/2$ drops by at least half on each recursion level. The time bound follows immediately, and the total work is dominated by the work at the root of the recursion tree, which is $\tilde{O}(m/\epsilon^2)$. □

Finally, Theorem 7.1.3 follows from Corollaries 7.3.2 and 7.3.10, and Theorem 7.1.1 and Theorem 7.1.2 follow from the addition of Theorem 7.3.7 and Corollary 7.3.6, respectively.

7.3.1 ℓ_1-Embedding from Approximate SSSP Potential

In this section, we briefly overview our ℓ_1-embedding algorithm, which is necessary for Theorem 7.3.1 and hence, the reduction from ℓ_1-embedding to smaller instances of approximate SSSP potentials. Our ℓ_1-embedding algorithm is very similar to Bourgain's embedding as presented in [74], except utilizing approximate SSSP instead of exact, as well as slightly simplified at the expense of several logarithmic factors. Due to its similarily, we defer its proof to Section 7.10.

Theorem 7.3.11

Let $G = (V, E)$ be a graph with n vertices and m edges, and let $\mathcal{A}$ be an algorithm that inputs any vertex set $S \subseteq V$ and outputs a $(1 + 1/\log n)$-approximate S-SSSP potential of G. Then, there is an algorithm that computes an ℓ_1-embedding of G into $O(\log n)$ dimensions with distortion $O(\log^{4.5} n)$ and calls $\mathcal{A}$ at most $O(\log^2 n)$ times, plus $\tilde{O}(m)$ additional work and polylog(n) additional time.

We will focus our attention on a slightly different variant which we show implies Theorem 7.3.11:

Lemma 7.3.12

Let $G = (V, E)$ be a graph with n vertices and m edges, and let $\mathcal{A}$ be an algorithm that inputs any vertex set $S \subseteq V$ and outputs a $(1 + 1/\log n)$-approximate S-SSSP potential of G. Then, there is an algorithm that computes an ℓ_1-embedding of G into $O(\log^2 n)$ dimensions with distortion $O(\log^3 n)$ and calls $\mathcal{A}$ at most $O(\log^2 n)$ times, plus $\tilde{O}(m)$ additional work and polylog(n) additional time.

Lemma 7.3.12 is proved in Section 7.10.2. We now show that Lemma 7.3.12 implies Theorem 7.3.11. Since the ℓ_1 and ℓ_2 metrics are at most a multiplicative $\sqrt{k}$ factor apart in dimension k, the embedding of Lemma 7.3.12 has distortion $O(\log^3 n) \cdot \sqrt{O(\log^2 n)} =$

$O(\log^4 n)$ in the ℓ_2 metric. Next, apply Johnson-Lindenstrauss dimensionality reduction [52] on this set of vectors, reducing the dimension to $O(\log n)$ with a constant factor increase in the distortion. We now move back to the ℓ_1 metric, incurring another $O(\sqrt{\log n})$ factor in the distortion, for a total of $O(\log^{4.5} n)$ distortion.

7.3.2 Sparsification and Recursion to Smaller Instances

In this section, we briefly overview the main ideas behind our sparsification process in order to reduce the ℓ_1-embedding problem to approximate SSSP instances of sufficiently smaller size:

Theorem: Restatement of Theorem 7.3.1

Let $G = (V, E)$ be a connected graph with n vertices and m edges with aspect ratio M, let $\beta \geq 1$ be a parameter, and let $\mathcal{A}$ be an algorithm that inputs (i) a connected graph on at most m/β vertices and edges with aspect ratio $\tilde{O}(\beta^2 M)$ and (ii) a source vertex s, and outputs a $(1 + 1/\log n)$-approximate s-SSSP potential. Then, there is an algorithm that computes an ℓ_1-embedding of G into $O(\log n)$ dimensions with distortion $O(\beta^2 \log^{6.5} n)$ and calls $\mathcal{A}$ at most $O(\log^2 n)$ times in parallel, plus $\tilde{O}(m)$ additional work and polylog(n) additional time.

One key tool we will use is the concept of *ultra-sparse spanners*, introduced by Elkin and Neiman [34]. Here, we will rename them to *ultra-spanners* to further emphasize their connection to *ultra-sparsifiers* in [63, 90]. These are spanners that are so sparse that they are almost "tree-like" when the graph is sparse enough: a graph with $(n-1)+t$ edges for some small t (say, $t = m/\text{polylog}(n)$). We will utilize the following ultra-spanner construction, which is adapted from the one of [82]; while theirs is not ultra-sparse, we modify it to be, at the expense of an additional k factor in the stretch. The ultra-spanner algorithm is deferred to Section 7.6.

Lemma 7.3.14

Given a weighted graph G with polynomial aspect ratio and a parameter $k \geq \Omega(1)$, there is an algorithm to compute a k^2-spanner of G with $(n-1) + O(\frac{m \log n}{k})$ edges in $\tilde{O}(m)$ work and polylog(n) time.

Why are ultra-spanners useful for us? Their key property, stated in the lemma below, is that we can compute an α-approximate SSSP potential on an ultra-spanner by recursively calling α-approximate SSSP potentials on a graph with potentially *much fewer vertices*. To develop some intuition on why this is possible, observe first that if a connected graph has $(n-1)$ edges, then it is a tree, and SSSP is very easy to solve on trees. If the graph has $(n-1)+t$ edges instead for some small value of t, then the graph is almost "tree-like" outside of at most $2t$ vertices: take an arbitrary spanning tree, and let these vertices be the endpoints of the t edges not on the spanning tree. We want to say that the graph is "easy" outside a graph on $2t$ vertices, so that we can solve a SSSP problem on the "hard" part of size $O(t)$ and then extend the solution to the rest of the graph in an efficient manner. This is indeed our approach, and it models closely off the concept of a j-tree by Madry [77], which is also used in Peng's recursive max-flow algorithm [90].

This recursion idea can be considered a *vertex-sparsification* step, following the edge-sparsification that the ultra-spanner achieves. We package the vertex-sparsification in the lemma below; while this lemma works for all t, the reader should imagine that $t = m/\text{polylog}(n)$, since that is the regime where the lemma will be applied. Due to its length and technical involvement, the proof is deferred to Section 7.5.

Lemma 7.3.15

Let $G = (V, E)$ be a connected graph with aspect ratio M with n vertices and $(n-1)+t$ edges, and let $\alpha > 0$ be a parameter. Let $\mathcal{A}$ be an algorithm that inputs a connected graph on at most $70t$ vertices and edges and aspect ratio $\tilde{O}(M)$ and outputs an α-approximate s-SSSP potential of that graph. Then, for any subset $S \subseteq V$, we can compute an α-approximate S-SSSP potential of G through a single call to $\mathcal{A}$, plus $\tilde{O}(m)$ additional work and $\text{polylog}(n)$ additional time.

We now prove Theorem 7.3.1 assuming Lemma 7.3.15:
Proof (Theorem 7.3.1). Invoke Lemma 7.3.14 with $k := C\beta \log n$ for a large enough constant $C > 0$, producing a spanner H with $(n-1) + O(\frac{m \log n}{k})$ edges and stretch at most $k^2 = O(\beta^2 \log^2 n)$. Since H is a spanner, we have $\min_{u,v \in V} d_H(u, v) \geq \min_{u,v \in V} d_G(u, v)$ and $\max_{u,v \in V} d_H(u, v) \leq k^2 \max_{u,v \in V} d_G(u, v)$, so H has aspect ratio $\tilde{O}(\beta^2 M)$. Since G is connected, we have $O(m \log n/k) \leq m/(70\beta)$ for C large enough, so H has at most $(n-1) + m/(70\beta)$ edges. Then, apply Lemma 7.3.15 on H with $t := m/(70\beta)$, $\alpha := 1 + 1/\log n$, and the algorithm $\mathcal{A}$, producing an algorithm $\mathcal{A}_H$ that inputs any vertex set $S \subseteq V$ and outputs an $(1 + 1/\log n)$-approximate S-SSSP potential on H through a single call to $\mathcal{A}$, plus $\tilde{O}(m)$ additional work and $\text{polylog}(n)$ additional time.

Next, apply Theorem 7.3.11 on the spanner H with algorithm $\mathcal{A}_H$, embedding H into $O(\log n)$ dimensions with distortion $O(\log^{4.5} n)$ through $O(\log^2 n)$ calls to $\mathcal{A}_H$, which in turn makes $O(\log^2 n)$ calls to $\mathcal{A}$; the additional work and time remain $\tilde{O}(m)$ and $\text{polylog}(n)$, respectively.

Finally, since H is a spanner for G with stretch $O(\beta^2 \log^2 n)$, the ℓ_1-embedding of H with stretch $O(\log^{4.5} n)$ is automatically an ℓ_1-embedding of G with distortion $O(\beta^2 \log^2 n) \cdot O(\log^{4.5} n) = O(\beta^2 \log^{6.5} n)$. □

7.4 ℓ_1-Oblivious Routing and Sequential Transshipment

This section is dedicated to the sequential transshipment result (Theorem 7.1.4, restated below) and constitutes our main technical contribution of this chapter.

Theorem: Restatement of Theorem 7.1.4

There is an algorithm that, given an undirected graph with nonnegative weights and polynomial aspect ratio, computes a $(1+\epsilon)$-approximation to minimum transshipment in time $O((m\log^{10} n + n\log^{15} n)\,\epsilon^{-2}\,(\log\log n)^{O(1)})$.

Throughout the section, we make no references to parallel algorithms, keeping all our algorithms entirely sequential in an effort to focus solely on Theorem 7.1.4. Nevertheless, to a reader with parallel algorithms in mind, it should be clear that all algorithms in this section can be parallelized to require $\text{polylog}(n)$ parallel time. To streamline the transition to parallel algorithms in the rest of this chapter, we package a parallel version of the main routine in this section in an easy-to-use statement, Corollary 7.3.3.

7.4.1 Improved ℓ_1-Oblivious Routing: Our Techniques

The key technical ingredient in our transshipment algorithm is an improved ℓ_1-oblivious routing, scheme. Our algorithm begins similarly to Sherman's [97]: compute an ℓ_1-embedding into low dimensions at a small loss in approximation. Sherman chooses dimension $O(\sqrt{\log n})$ and loses a $2^{O(\sqrt{\log n})}$ factor in the distortion, and then constructs an oblivious routing in the embedded space in time exponential in the dimension. Our oblivious routing is polynomial in the dimension, so we can afford to choose dimension $O(\log n)$, giving us $\text{polylog}(n)$ distortion. The benefit in the ℓ_1-embedding is that we now have a nice geometric property of the vertices, which are now points in $O(\log n)$-dimensional space under the ℓ_1 metric.

At this point, let us provide some intuition for the oblivious routing problem in ℓ_1 space. Suppose for simplicity that the dimension is 1 (i.e., we are on the real line) and that all vertices have integer coordinates. That is, every vertex $v \in V$ is now an integer on the real line, i.e., $V \subseteq \mathbb{Z}$. We will now (informally) define the problem of oblivious routing on the line:[3]

1. Our input is a set of points $V \subseteq \mathbb{Z}$. There is also a function $b : V \to \mathbb{R}$ of demands with $\sum_{v\in V} b(v) = 0$ that is *unknown* to us.
2. On each step, we can choose any two points $x, y \in \mathbb{Z}$ and a scalar $c \in \mathbb{R}$, and "shift" c times the demand at x to location y. That is, we simultaneously update $b(x) \leftarrow b(x) - c\cdot b(x)$ and $b(y) \leftarrow b(y) + c\cdot b(x)$. We pay $c\cdot b(x)\cdot|x-y|$ total cost for this step. Again, we do not know how much we pay. Let an *iteration* be defined as one or more such steps executed in parallel.
3. After a number of iterations, we declare that we are done. At this point, we *must* be certain that the demand is 0 everywhere: $b(x) = 0$ for all $x \in \mathbb{Z}$.
4. Once we are done, we learn the set of initial demands, sum up our total cost, and

[3]Our formal definition of oblivious routing is in matrix notation, and is considerably less intuitive. Therefore, we hope to present enough of our intuition in this section.

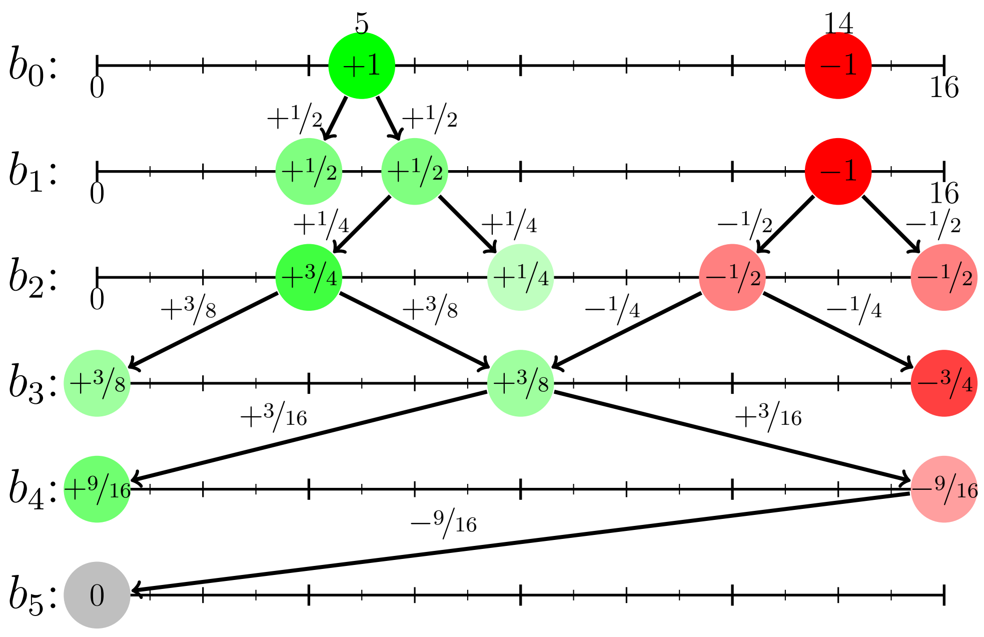

Figure 7.2: Oblivious routing in 1-dimensional (ℓ_1-) space. Here, there are only two locations with nonzero demand at the beginning: +1 demand at point 5 and −1 demand at point 14. The optimal routing for each b_t has cost 18, and the routing costs of iterations $t = 1, 2, 3, 4, 5$ are 1, 3, 5, 3, and 9, respectively.

compare it to the optimal strategy we could have taken *if we had known the demands beforehand*. We would like our cost to be comparable with this retrospective optimum. In particular, we would like to pay at most polylog(n) times this optimum.

We maintain functions $b_0, b_1, b_2 \ldots : \mathbb{Z} \to \mathbb{R}$ that track how much demand remains at each (integer) point after each iteration. Given a demand vector (function) $b : V \to \mathbb{R}$, every vertex $v \in V$ has an initial demand $b_0(v) := b(v)$, and these demands sum to 0. Consider the following oblivious routing algorithm: for each iteration $t = 1, 2, \ldots$, every point $x \in \mathbb{Z}$ with $x \equiv 2^{t-1} \bmod 2^t$ sends $b_t(x)/2$ flow to point $x - 2^{t-1}$ and $b_t(x)/2$ flow to point $x + 2^{t-1}$; let $b_{t+1}(x)$ be the new set of demands (see Figure 7.2).

This is actually Sherman's oblivious routing in 1-dimensional space. He proves the following two properties of the routing:

1. After each iteration t, the optimal routing for the remaining points can never increase. (In Figure 7.2, the optimal routing of each b_t is exactly 9.)
2. The routing cost at each iteration t is at most the optimal cost of routing b_t. (In Figure 7.2, the routing costs of iterations $t = 1, 2, 3, 4$ are 1, 3, 5, and 3, respectively.)

Let us assume that $V \subseteq [0, 1, 2, \ldots, n^c]$ for some constant c, that is, all points in V are nonnegative, polynomial-sized integers. Then, after $\lceil \log_2(n^c) \rceil = O(\log n)$ iterations, all points are either on 0 or $2^{\lceil \log_2(n^c) \rceil}$. Thus, moving all demand from 0 to $2^{\lceil \log_2(n^c) \rceil}$ finishes the oblivious routing. From the two properties above, this oblivious routing can be shown to be

$O(\log n)$-competitive.

We believe this simple scheme provides a good intuition of what an oblivious routing algorithm requires. In particular, it must be *unbiased*, in that demand from a given vertex must be spread evenly to the left and right. This is because we do not know where the demands lie, so our best bet is to spread equal amounts of demand left and right.

Sherman's oblivious routing extends this idea to higher dimensions. The actual routing is more complicated to describe, but as an example, on iteration $t = 1$, a point $x = (1, 1, 1, \dots, 1)$ will need to send $b(x)/2^k$ flow to each of the 2^k points in $\{0, 2\}^k$. In other words, the running time can be exponential in the dimension.

This is where our oblivious routing algorithm deviates from Sherman's. To avoid the issue of sending flow to too many other points, we make use of random sampling: on each iteration, every point sends its flow to polylog(n) randomly chosen points close-by. These random points need to be correlated sufficiently well so that we can control the total number of points. (In particular, we do not want the number of points to increase by factor $O(\log n)$ each iteration, which would happen on a naive attempt.)

To solve this issue, we use the concept of *randomly shifted grids* popular in low-dimensional computational geometric algorithms [49]: overlay a randomly shifted grid of a specified size W in the $\mathbb{R}^k$-dimensional space. Every point sends a fraction of its demand to (say) the midpoint of the grid cell containing it.[4] The benefit in grid shifting is that many nearby points can coalesce to the same midpoints of a grid, controlling the growth of the number of points. We compute $s =$ polylog(n) such grids, with each point sending $1/s$ fraction to the midpoint of each grid; this is to control the variance, so that we can apply concentration bounds to show that we are still approximately unbiased from each point.

7.4.2 Sherman's Framework

Below, we state a paraphrased version of Sherman's framework [97]. For the simplest reference, see Corollary 1 and Lemma 4 of [60]. We also provide a proof via *multiplicative weights* in Section 7.7, whose running time suffers an additional factor of $\log(n/\epsilon)$ due to a binary search overhead. For Theorem 7.1.4, we will use the theorem below, while for the parallel algorithms, the weaker Theorem 7.7.1 suffices.

[4]For notational simplicity, our algorithm will actually send to the "lower-left" corner of each grid, but for this section, midpoint is more intuitive to think about.

Theorem 7.4.2: Sherman, paraphrased

Given a transshipment problem, suppose we have already computed a matrix R satisfying:

1. For all demand vectors $b \in \mathbb{R}^n$, $\mathsf{opt}(b) \le \mathbb{1}Rb \le \kappa \cdot \mathsf{opt}(b)$
2. Matrix-vector products with R and R^T can be computed in M time

Then, for any transshipment instance with demand vector b, we can compute a flow vector $\tilde{f}$ and a vector of potentials $\tilde{\phi}$ in $O(\kappa^2(m+n+M)\log(m)(\epsilon^{-2}+\log(1/\beta)))$ sequential time that satisfies:

1. $\left\|C\tilde{f}\right\|_1 \le (1+\epsilon)b^T\tilde{\phi} \le (1+\epsilon)\,\mathsf{opt}(b)$
2. $\mathsf{opt}(A\tilde{f}-b) \le \beta\,\mathsf{opt}(b)$

The matrix R encodes the oblivious routing algorithm. Also, intuitively, the more efficient the oblivious routing, the sparser the matrix R, although this relation is not as well-defined. Nevertheless, there is an equivalence between oblivious routing schemes and matrices R that satisfy requirement (1) of Theorem 7.4.2. But since Sherman's framework uses steepest descent methods that involve matrix algebra, a matrix R with efficient matrix-vector multiplications is most convenient for the framework.

Our main technical result is computing such a matrix R efficiently:

Theorem 7.4.3: Computing R

Given a transshipment problem, we can compute a matrix R with $O(n\log^5 n(\log\log n)^{O(1)})$ nonzero entries, such that for any demand vector b,

$$\mathsf{opt}(b) \le \mathbb{1}Rb \le O(\log^{4.5} n)\cdot \mathsf{opt}(b).$$

The algorithm succeeds w.h.p., and runs in $O(m\log^2 n + n\log^{10} n(\log\log n)^{O(1)})$ sequential time.

With this fast routing algorithm in hand, our main theorem, Theorem 7.1.4, follows immediately. Our proof uses low-stretch spanning trees [4], so for a self-contained rendition, we remark after the proof that low-stretch spanning trees can be removed at the expense of another $\log n$ factor.

Proof of Theorem 7.1.4. Apply Theorem 7.4.2 with the parameters $\kappa := O(\log^{4.5} n)$ and $M := O(n\log^5 n(\log\log n)^{O(1)})$ guaranteed by Theorem 7.4.3, along with $\beta := \Theta(\epsilon/(\log n\log\log n))$. This takes time

$$O(\log^9 n\cdot(m+n\log^5 n)\cdot\log n\cdot\epsilon^{-2}\cdot(\log\log n)^{O(1)}) = O((m\log^{10} n+n\log^{15} n)\epsilon^{-2}(\log\log n)^{O(1)}),$$

and outputs a flow $\tilde{f}$ with $\mathbb{1}Cf \le (1+\epsilon)\,\mathsf{opt}(b)$ and $\mathsf{opt}(A\tilde{f}-b) \le \beta\,\mathsf{opt}(b)$.

To route the remaining demand $A\tilde{f}-b$, for $O(\log n)$ independent iterations, compute a low-stretch spanning tree in $O(n \log n \log\log n)$ time with expected stretch $O(\log n \log\log n)$ [4] and solve (exact) transshipment in linear time on the tree. In each iteration, the expected cost is at most $O(\log n \log\log n) \cdot \beta\, \mathsf{opt}(b) = \epsilon\, \mathsf{opt}(b)$ for an appropriate choice of β, so w.h.p., one iteration has cost at most twice the expectation. Let f' be this flow, which satisfies $\mathbb{1}Cf' \leq 2\epsilon\, \mathsf{opt}(b)$ and $Af' = A\tilde{f} - b$. The composed flow $\tilde{f} - f'$ is our final flow, which satisfies $\mathbb{1}C(f - f') \leq \mathbb{1}Cf + \mathbb{1}Cf' \leq (1 + 3\epsilon)\, \mathsf{opt}(b)$ and $A(\tilde{f} - f') = b$. Finally, to obtain a $(1+\epsilon)$-approximation, we can simply reset $\epsilon \leftarrow \epsilon/3$. $\square$

Remark 7.4.4

To eliminate the use of low-stretch spanning trees, we can set $\beta := \epsilon/n$ instead, picking up another $\log n$ factor in Theorem 7.4.2. Then, we can route the remaining demand along a minimum spanning tree, which is an $(n-1)$-approximation of optimum, or at most $(n-1)\, \beta \mathsf{opt}(b) \leq \epsilon\, \mathsf{opt}(b)$.

7.4.3 Polynomial Aspect Ratio

Throughout this section, we assume that the input graph has polynomial aspect ratio, since that is assumed in Theorem 7.1.4.

7.4.4 Reduction to ℓ_1 Metric

The reduction to the ℓ_1 metric is standard, via Bourgain's embedding:

Definition 7.4.5

For $p \geq 1$, an *ℓ_p-embedding* of a graph $G = (V, E)$ with *distortion* α and *dimension* k is a collection of vectors $\{x_v \in \mathbb{R}^k : v \in V\}$ such that

$$\forall u, v \in V : \; d_G(u, v) \leq \|x_u - x_v\|_p \leq \alpha\, d_G(u, v).$$

Theorem 7.4.6: Fast Bourgain's embedding

Given a graph with m edges, there is a randomized $O(m \log^2 n)$ time algorithm that computes an ℓ_1-embedding of the graph with distortion $O(\log^{1.5} n)$ and dimension $O(\log n)$.

Proof (Sketch). Apply the fast embedding algorithm of [74] in ℓ_2, which runs in $O(m \log^2 n)$ randomized time and w.h.p., computes an ℓ_2-embedding of the graph with distortion $O(\log n)$ and dimension $O(\log^2 n)$. Next, apply Johnson-Lindenstrauss dimensionality reduction [52] on this set of vectors, reducing the dimension to $O(\log n)$ with a constant factor increase in the distortion. Lastly, since the ℓ_1 and ℓ_2 metrics are at most a multiplicative $\sqrt{k}$ factor apart in dimension k, this same set of vectors in $O(\log n)$ dimensions has distortion $O(\log^{1.5} n)$ in the ℓ_1 metric. □

Finally, since our input graph is assumed to have polynomial aspect ratio, so do the embedded points under the ℓ_1 metric. In particular, suppose that before applying Theorem 7.4.6 we scaled the graph G so that the smallest edge had length 1. Then, the embedding satisfies the following:

Assumption 7.4.7: Polynomial aspect ratio

For some constant $c > 0$, the vectors $\{x_v : v \in V\}$ satisfy $1 \le d(u, v) \le n^c$ for all $u, v \in V$.

7.4.5 Oblivious Routing on ℓ_1 Metric

In this section, we work under the ℓ_1 metric in $O(\log n)$ dimensions (the setting established by Theorem 7.4.6) with the additional Assumption 7.4.7. Our main technical result is:

Theorem 7.4.8

We can compute a matrix R with $O(n \log^5 n (\log\log n)^{O(1)})$ nonzero entries, such that for any demand vector b,

$$\mathsf{opt}(b) \le \mathbb{1}Rb \le O(\log^3 n) \cdot \mathsf{opt}(b).$$

The algorithm succeeds w.h.p., and runs in $O(n \log^{10} (\log\log n)^{O(1)})$ sequential time.

Together with the $O(\log^{1.5} n)$ additional distortion from Theorem 7.4.6, this proves Theorem 7.4.3.

Before we begin with the algorithm, we first make a reduction from "w.h.p., for all b" to the weaker statement "for each b, w.h.p.". The former requires that w.h.p., the statement holds for *every* demand vector b, while the latter requires that for any *given* demand vector b, the statement holds w.h.p. (Since there are uncountably many such b, the latter does not imply the former in general.) This simplifies our argument, since we only need to focus on a given demand vector b, which will often be fixed throughout a section. Before we state and prove the reduction, for each $v \in V$, let us define $\chi_v : V \to \mathbb{R}$ as the function that is 1 at v and 0 elsewhere.

Lemma 7.4.9

Suppose a randomized algorithm outputs a matrix R such that for any given demand vector b, we have $\mathsf{opt}(b) \le \mathbb{1}Rb$ with probability 1, and $\mathbb{1}Rb \le \kappa \cdot \mathsf{opt}(b)$ w.h.p. Then, this same matrix R satisfies the following stronger property: w.h.p., for any demand vector b, we have $\mathsf{opt}(b) \le \mathbb{1}Rb \le \kappa \cdot \mathsf{opt}(b)$.

Proof. W.h.p., the matrix R satisfies $\mathbb{1}R(\chi_u - \chi_v) \le \kappa \cdot \mathsf{opt}(\chi_u - \chi_v)$ for each of the $O(n^2)$ demand vectors $\chi_u - \chi_v$ $(u, v \in V)$. We claim that in this case, R actually satisfies $\mathsf{opt}(b) \le \mathbb{1}Rb \le \kappa \cdot \mathsf{opt}(b)$ for all demand vectors b.

Fix any demand vector b, and suppose that the flow achieving $\mathsf{opt}(b)$ routes $f(u,v) \ge 0$ flow from u to v for each $u, v \in \mathbb{R}^k$, so that $b = \sum_{u,v} f(u,v) \cdot (\chi_u - \chi_v)$ and $\mathsf{opt}(b) = \sum_{u,v} f(u,v) \cdot \mathbb{1}u - v$. Then, we still have $\mathsf{opt}(b) \le \mathbb{1}Rb$ by assumption, and for the other direction, we have

$$\begin{aligned}\mathbb{1}Rb = \mathbb{1}R \cdot \sum_{u,v} f(u,v)(\chi_u - \chi_v) &\le \sum_{u,v} f(u,v)\mathbb{1}R(\chi_u - \chi_v) \\ &\le \kappa \cdot \sum_{u,v} f(u,v)\,\mathsf{opt}(\chi_u - \chi_v) = \kappa \cdot \mathsf{opt}(b),\end{aligned}$$

as desired. □

We also introduce a specific formulation of a *routing* that helps in the analysis of our algorithm:

Definition 7.4.10: Routing

Given a metric space (V, d), a *routing* is a function $R : V \times V \to \mathbb{R}$ such that

$$\forall u, v \in V: \ R(u,v) = -R(v,u).$$

A routing R *satisfies* demand vector $b \in \mathbb{R}^V$ if

$$\forall v \in V: \sum_{u \in V} R(u,v) = b_v.$$

A routing R has *cost*

$$\mathsf{cost}(R) := \sum_{u,v \in V} |R(u,v)| \cdot \mathbb{1}u - v,$$

and is *optimal* for demand vector b if it minimizes $\mathsf{cost}(R)$ over all routings R' satisfying b.

For example, if $b = \chi_u - \chi_v$ for some $u, v \in V$, then one feasible routing (in fact, the optimal one) is $R(u,v) = 1$, $R(v,u) = -1$, and $R(x,y) = 0$ for all other pairs (x,y), which has cost $2\mathbb{1}u - v$.

Note that $\mathsf{cost}(R)$ is actually *twice* the value of the actual transshipment cost in the ℓ_1 metric. However, since this notion of routing is only relevant in our analysis, and we are suffering a polylog(n) approximation anyway, we keep it this way for future simplicity.

Observation 7.4.11

Given a metric space (V, d) and demand vector $b \in \mathbb{R}^V$, $\mathsf{opt}(b)$ is equal to the minimum value of $\frac{1}{2}\mathsf{cost}(R)$ over all routings R that satisfy demand vector b.

We first introduce our algorithm in pseudocode below, along with the following notations. For real numbers x and $W > 0$, define $\lfloor x \rfloor_W := \lfloor x/W \rfloor \cdot W$ as the greatest (integer) multiple of W less than or equal to W (so that if $W = 1$, then $\lfloor x \rfloor_W = \lfloor x \rfloor$), and similarly, define $\lceil x \rceil_W := \lceil x/W \rceil \cdot W$ as the smallest (integer) multiple of W greater than or equal to W.

The lines marked *imaginary* are actually not executed by the algorithm. They are present to define the "imaginary" routings R_t^*, which exist only for our analysis. We could have defined the R_t^* separately from the algorithm, but we decided that including them alongside the algorithm is more concise and (more importantly) illustrative.

Lastly, we remark that the algorithm does not require the input $b \in \mathbb{R}^V$ to be a demand vector. This observation is important when building the matrix R, where we will evaluate the algorithm on only the vectors χ_v for $v \in V$, which are not demand vectors.

Proof Outline. The purpose of the "imaginary" routing R_t^* is to upper bound our actual cost. For R_t^* to be a reasonable upper bound, it should not increase too much over the iterations. These properties are captured in the two lemmas below.

Lemma 7.4.12

The total cost of routing on each iteration t (lines 17 and 18) is at most $kw \cdot \mathsf{cost}(R_t^*)$.

Lemma 7.4.13

With probability $1 - n^{-\omega(1)}$, $\mathsf{cost}(R_{t+1}^*) \le (1 + \frac{1}{\log n})\,\mathsf{cost}(R_t^*)$ for each iteration t.

The last routing on lines 24 and 25 is handled by the following lemma.

Lemma 7.4.14

The total cost of routing on lines 24 and 25 is at most $O(kw) \cdot \mathsf{cost}(R_{T+1}^*)$.

The three lemmas above imply the following corollary:

Corollary 7.4.15: Cost of routing

With probability $1 - n^{-\omega(1)}$, the total cost of routing in the algorithm is at most $O(kwT) \cdot \mathsf{opt}(b)$.

Algorithm 9 Routing(V, b)

Input:
(1) V, a set of n vectors in $\mathbb{R}^k$ satisfying Assumption 7.4.7, where $k = O(\log n)$
(2) $b \in \mathbb{R}^V$, a (not necessarily demand) vector

```
1: Initialize $w \leftarrow \lceil \log n \rceil$, $s \leftarrow \lceil \log^4 n \log\log n \rceil$, $T \leftarrow \lceil \log_w(n^c) \rceil$
2: Initialize $V_0 \leftarrow V$
3: Initialize function $b_0 : \mathbb{R}^V \to \mathbb{R}$ satisfying $b_0(v) = b_v$ for all $v \in V$ and $b(x) = 0$ for all $x \notin V$
4: Initialize function $R : \mathbb{R}^V \times \mathbb{R}^V \to \mathbb{R}$ as the zero function (i.e., $R(x, y) = 0$ for all $x, y \in \mathbb{R}^V$)
5: Initialize $R_0^* : \mathbb{R}^V \times \mathbb{R}^V \to \mathbb{R}$ as the optimal routing satisfying $b_0$    ▷ Imaginary
6: for iteration $t = 0, 1, 2, \ldots, T$ do
7:     $W \leftarrow w^t$, a positive integer
8:     Initialize $V_{t+1} \leftarrow \emptyset$
9:     Initialize $b_{t+1} : \mathbb{R}^V \to \mathbb{R}$ as the zero function
10:    Initialize $R_{t+1}^* : \mathbb{R}^V \times \mathbb{R}^V \to \mathbb{R}$ as the zero function    ▷ Imaginary
11:    for independent trial $j = 1, 2, \ldots, s$ do
12:        Choose independent, uniformly random real numbers $r_1, \ldots, r_k \in [0, W)$
13:        Define $h_j : \mathbb{R}^k \to \mathbb{R}^k$ as $(h_j(x))_i = \lfloor x_i + r \rfloor_W$ for all $i \in [k]$
14:        for $x \in V_t : b_t(x) \neq 0$ do
15:            $y \leftarrow h_j(x)$
16:            $V_{t+1} \leftarrow V_{t+1} \cup \{y\}$    ▷ $V_{t+1}$ is the set of points with flow after iteration $t$
17:            $R(x, y) \leftarrow R(x, y) + b_t(x)/s$    ▷ Send $b_t(x)/s$ flow from $x$ to $y$
18:            $R(y, x) \leftarrow R(y, x) - b_t(x)/s$
19:            $b_{t+1}(y) \leftarrow b_{t+1}(y) + b_t(x)/s$
20:        for $(x, y) \in \mathbb{R}^k \times \mathbb{R}^k : R_t^*(x, y) \neq 0$ do    ▷ Imaginary
21:            $R_{t+1}^*(h_j(x), h_j(y)) \leftarrow R_{t+1}^*(h_j(x), h_j(y)) + R_t^*(x, y)/s$    ▷ Imaginary: move $1/s$ fraction flow
22: Let $y \in V_{T+1}$ be arbitrary
23: for $x \in V_{T+1} : b_{T+1}(x) \neq 0$ do
24:     $R(x, y) \leftarrow R(x, y) + b_{T+1}(x)$    ▷ Route all demand in $V_{T+1}$ to arbitrary vertex $y$ in $V_{T+1}$
25:     $R(y, x) \leftarrow R(y, x) - b_{T+1}(x)$
```

Proof. By applying Lemma 7.4.13 inductively over all t, with probability $1 - n^{-\omega(1)}$,

$$\begin{aligned}\mathsf{cost}(R^*_{t+1}) \le \left(1 + \frac{1}{\log n}\right)^t \mathsf{cost}(R^*_0) = \left(1 + \frac{1}{\log n}\right)^t \mathsf{opt}(b) \le \left(1 + \frac{1}{\log n}\right)^{O(\log n)} \mathsf{opt}(b) \\ = O(1) \cdot \mathsf{opt}(b).\end{aligned}$$

By Lemma 7.4.12, the cost of routing on iteration t is at most $kw \cdot \mathsf{cost}(R^*_t) \le O(kw) \cdot \mathsf{opt}(b)$. Summing over all t, we obtain a total cost of $O(kwT) \cdot \mathsf{opt}(b)$ over iterations 0 through T. Finally, by Lemma 7.4.14, the cost of routing on lines 24 and 25 is at most $O(kw) \cdot \mathsf{cost}(R^*_{T+1}) \le O(kw) \cdot \mathsf{opt}(b)$ as well. □

At the same time, the routing should be "sparse", to allow for a near-linear time algorithm. Our sparsity is captured by the following lemma.

Lemma 7.4.16

For each χ_v, if we run Algorithm 9 on demands χ_v, every function b_t has $O(s)$ nonzero values in expectation. Moreover, each function b_t can be computed in $O(s^2)$ expected time.

This sparsity guarantee ensures that the matrix R that we compute is also sparse, specified in the lemma below.

Lemma 7.4.17

We can compute a matrix R such that $\mathbb{1}Rb$ approximates the cost of routing in Algorithm 9 to factor $O(1)$, and R has $O(sTn) = O(n \log^5 n (\log\log n)^{O(1)})$ nonzero entries. The algorithm succeeds w.h.p., and runs in time $O(s^2 Tn \log n) = O(n \log^{10} n (\log\log n)^{O(1)})$.

Finally, with Corollary 7.4.15 and Lemmas 7.4.16 and 7.4.17, we prove Theorem 7.4.8 below:
Proof of Theorem 7.4.8. By Lemma 7.4.17, we can compute a matrix R that approximates the cost of routing in Algorithm 9 to factor $O(1)$. By Corollary 7.4.15, this cost of routing is at most $O(kwT) \cdot \mathsf{opt}(b)$, and it is clearly at least $\mathsf{opt}(b)$. Thus, R approximates $\mathsf{opt}(b)$ by an $O(kwT) = O(\log^3 n)$ factor. The requirements on R are guaranteed by Lemma 7.4.17. □

Proof of Approximation (Lemmas 7.4.12 and 7.4.13). We first begin with a few invariants of Algorithm 9, whose proofs are trivial by inspection:

Invariant 7.4.18

At the end of iteration t, R satisfies demand vector b_{t+1}.

Proof. Suppose by induction on t that R satisfies demand vector b_t at the beginning of iteration t. Recall that for R to satisfy b_{t+1} at the end of iteration t, we must have

$\sum_u R(u,v) = b_{t+1}(v)$ for all v by then. For each v, we track the change in $\sum_u R(u,v)$ and show that the total change on iteration t is exactly $b_{t+1}(v) - b_t(v)$, which is sufficient for our claim. By lines 17 and 18, for each x satisfying $b_t(x) \neq 0$ and each $j \in [s]$, the value $\sum_u R(u, h_j(x))$ increases by $b_t(x)/s$ and the value $\sum_u R(u,x)$ decreases by $b_t(x)/s$. For each x with $b_t(x) \neq 0$, the s decreases add up to a total of $b_t(x)$. As for $b_{t+1}(v) - b_t(v)$, a demand of $b_t(v)$ is not transferred over to $b_{t+1}(v)$ if $b_t(v) \neq 0$, and $b_{t+1}(v)$ increases by $b_t(x)/s$ for each x, j with $h_j(x) = v$. Altogether, the differences in $\sum_u R(u,v)$ and $b_{t+1}(v) - b_t(v)$ match. □

Invariant 7.4.19

R^*_{t+1} satisfies demand vector b_{t+1}.

Proof. Suppose by induction on t that R^*_t satisfies demand vector b_t; the base case $t = 0$ is trivial. For each v satisfying $b_t(v) \neq 0$ and each $j \in [s]$, the value $\sum_u R^*_{t+1}(u, h_j(v))$ increases by $\sum_{x,v} R^*_t(x,v)/s$ (line 21), which by induction is exactly $b_t(v)/s$. This matches the increase of $b_{t+1}(h_j(v))$ by $b_t(v)/s$ on line 19. □

Invariant 7.4.20

For each pair (x,y) with $R^*_{t+1}(x,y) \neq 0$, $x - y$ has all coordinates an (integral) multiple of w^t.

Proof. The only changes to R^*_{t+1} are the $R^*_{t+1}(h_j(x), h_j(y))$ changes on line 21. By definition of h_j, we have that $h_j(u) - h_j(v)$ is a multiple of $W = w^t$ for all u, v. □

Lemma: Restatement of Lemma 7.4.12

The total cost of routing on each iteration t (lines 17 and 18) is at most $kw \cdot \mathsf{cost}(R^*_t)$.

Proof. For each trial $j \in [s]$, by construction of $h_j(x)$ (line 13), we have $\mathbb{1}(h_j(x))_i - x_i \le kW$, which incurs a cost of at most $|b_t(x)/s| \cdot kW$ in the routing R (lines 17 and 18). Over all s iterations, each $x \in \mathbb{R}^k$ with $b_t(x) \neq 0$ is responsible for at most $|b_t(x)| \cdot kW$ cost.

We now bound $\sum_x |b_t(x)| \cdot kW$ in terms of $\mathsf{cost}(R^*_t)$. By Invariant 7.4.19, for each x with $b_t(x) \neq 0$,

$$\sum_y |R^*_t(x,y)| \ge \Big| \sum_y R^*_t(x,y) \Big| = |b_t(x)|.$$

(Here, the summation is over the finitely many y that produce a nonzero summand.) Summing over all such x, we get

$$\sum_{x:b_t(x)\neq 0} |b_t(x)| \le \sum_{x:b_t(x)\neq 0} \sum_y |R^*_t(x,y)| \le \sum_{x,y} |R^*_t(x,y)|. \tag{7.2}$$

By Invariant 7.4.20, we have $\mathbb{1}x - y \geq w^{t-1}$ for each (x, y) with $R_t^*(x, y) \neq 0$. Therefore,

$$\mathsf{cost}(R_t^*) = \sum_{x,y} |R_t^*(x, y)| \cdot \mathbb{1}x - y \geq \sum_{x,y} |R_t^*(x, y)| \cdot w^{t-1}. \tag{7.3}$$

Thus, the cost is at most

$$\sum_x |b_t(x)| \cdot kW \overset{(7.2)}{\leq} \sum_{x,y} |R_t^*(x, y)| \cdot kW = kw \cdot \sum_{x,y} |R_t^*(x, y)| \cdot w^{t-1} \overset{(7.3)}{\leq} kw \cdot \mathsf{cost}(R_t^*). \tag{7.4}$$

□

Claim 7.4.22

For each $t \in [T + 1]$, R_t^* has support size $n^{O(1)}$.

Proof. For each $t \in [0, T]$, by lines 20 and 21, every (x, y) with $R_t^*(x, y)$ is responsible for creating at most $s \leq O(\log^5 n)$ nonzero values in R_{t+1}^*. Also, R_0^* is supported in V, so it has support size $n^{O(1)}$. Therefore, R_t^* has support size at most

$$n^{O(1)} \cdot s^{T+1} = n^{O(1)} \cdot (O(\log^5 n))^{O(\log n/ \log\log n)} = n^{O(1)}.$$

□

Lemma 7.4.23

Fix two points (x, y) with $R_{t+1}^*(x, y) \neq 0$, and fix a coordinate $i \in [k]$. With probability $1 - n^{-\omega(1)}$, we have

$$\frac{1}{s} \sum_{j=1}^{s} |(h_j(x))_i - (h_j(y))_i| \leq \left(1 + \frac{1}{\log n}\right) |x_i - y_i|. \tag{7.5}$$

Proof. Define $\delta_i := x_i - y_i$. First, if $\delta_i = 0 \iff x_i = y_i$, then $(h_j(x))_i = (h_j(y))_i$ with probability 1, so both sides of (7.5) are zero.

Assume now that $\delta_i > 0$. Throughout the proof, we recommend the reader assume $W = 1$ so that $\lfloor x \rfloor_W$ is simply $\lfloor x \rfloor$, etc., since the proof is unchanged upon scaling W. Define $\{x\}_W := x - \lfloor x \rfloor_W$, the "remainder" of x when divided by W.

Observe that for each of the s independent trials, $(h_j(x))_i - (h_j(y))_i = \lfloor \delta_i \rfloor_W$ with probability $1 - \{\delta_i\}_W/W$ and $(h_j(x))_i - (h_j(y))_i = \lceil \delta_i \rceil_W$ with probability $\{\delta_i\}_W/W$. In particular, $\mathbb{E}[(h_j(x))_i - (h_j(y))_i] = \delta_i$.

For $j \in [s]$, define random variable X_j as the value of $\big((h_j(x))_i - (h_j(y))_i - \lfloor \delta_i \rfloor_W\big)/W$ on the j'th independent trial, so that $X_j \in \{0, 1\}$ and $\mathbb{E}[X_j] = \{\delta_i\}_W/W$ for all j. We can express the LHS of (7.5) as

$$\begin{aligned}
\frac{1}{s}\sum_{j=1}^{s}|(h_j(x))_i-(h_j(y))_i| &= \frac{1}{s}\sum_{j=1}^{s}\big((h_j(x))_i-(h_j(y))_i\big)\\
&= \frac{1}{s}\sum_{j=1}^{s}\big(W\cdot X_j+\lfloor\delta_i\rfloor_W\big)\\
&= \frac{W}{s}\sum_{j=1}^{s}X_j+\lfloor\delta_i\rfloor_W\\
&= \frac{W}{s}\sum_{j=1}^{s}X_j+(\delta_i-\{\delta_i\}_W). \qquad (7.6)
\end{aligned}$$

Define $\mu := \sum_j \mathbb{E}[X_j] = (s/W)\,\{\delta_i\}_W$. By Invariant 7.4.20 applied to iteration $t-1$, we know that δ_i is a multiple of $w^{t-1} = W/w$, so $\{\delta_i\} \geq W/w$, which means $\mu \geq s/w$. Applying Chernoff bounds on the variables $X_1,\dots,X_s \in [0,1]$ with $\epsilon := 1/\log n$, we obtain

$$\Pr\left[\sum_{j=1}^{s}X_j \geq (1+\epsilon)\mu\right] \leq \exp(-\epsilon^2\mu/3) \leq \exp\left(-\frac{s}{3w\log^2 n}\right) = \exp(-\omega(\log n)) = n^{-\omega(1)}.$$

This means that with probability $1-n^{\omega(1)}$,

$$\begin{aligned}
\frac{1}{s}\sum_{j=1}^{s}|(h_j(x))_i-(h_j(y))_i| &\overset{(7.6)}{=} \frac{W}{s}\sum_{j=1}^{s}X_j+(\delta_i-\{\delta_i\}_W)\\
&\leq \frac{W}{s}(1+\epsilon)\mu+(\delta_i-\{\delta_i\}_W)\\
&= (1+\epsilon)\{\delta_i\}_W+\delta_i-\{\delta_i\}_W\\
&= \delta_i+\epsilon\{\delta_i\}_W\\
&\leq (1+\epsilon)\delta_i\\
&= \left(1+\frac{1}{\log n}\right)|x_i-y_i|,
\end{aligned}$$

completing (7.5).

Finally, for the case $\delta_i < 0$, we can simply swap x and y and use the $\delta_i > 0$ case. □

Proof of Lemma 7.4.13. By lines 20 and 21, every (x,y) with $R_t^*(x,y) \neq 0$ is responsible for a total cost of

$$\sum_{j=1}^{s}\frac{|R_t^*(x,y)|}{s}\cdot \mathbb{1}h_j(x)-h_j(y) = \frac{|R_t^*(x,y)|}{s}\sum_{j=1}^{s}\sum_{i=1}^{k}|(h_j(x))_i-(h_j(y))_i|.$$

We now take a union bound over all such (x, y) (at most $n^{O(1)}$ many by Claim 7.4.22). By Lemma 7.4.23, we have that with probability $1 - n^{-\omega(1)}$, the total cost is at most

$$\frac{|R_t^*(x,y)|}{s} \sum_{j=1}^{s} \sum_{i=1}^{k} \left(1 + \frac{1}{\log n}\right) |x_i - y_i| = \left(1 + \frac{1}{\log n}\right) |R_t^*(x,y)| \cdot \mathbb{1}x - y.$$

Summing over all such (x, y), we obtain $\mathsf{cost}(R_{t+1}^*) \leq (1 + \frac{1}{\log n})\, \mathsf{cost}(R_t^*)$, as desired. $\square$

Proof of Sparsity (Lemma 7.4.16). For the proof of Lemma 7.4.16, we first introduce the concept of a history graph that tracks the routed flow.

Definition 7.4.24

Define the *history graph* H to be the following digraph on vertex set $V(H) := (V_0 \times \{0\}) \cup (V_1 \times \{1\}) \cup \cdots \cup (V_{T+1} \times \{T+1\})$. For every $t \in \{0, 1, \ldots, T\}$ and every x, y such that line 17 is executed at least once on $R(x, y)$, add a directed edge $((x, t), (y, t+1))$ in H. (By Invariant 7.4.25 below, every such x, y must satisfy $x \in V_t$ and $y \in V_{t+1}$.) A vertex $(x, t) \in V(H)$ *originates* from vertex $v \in V = V_0$ if there is a directed path in H from $(v, 0)$ to (x, t).

Invariant 7.4.25

For each x with $b_{t+1}(x) \neq 0$, we have $x \in V_{t+1}$.

Proof. Every $x \in V$ with value $b_{t+1}(x)$ modified in line 19 is added into V_{t+1} in line 17. $\square$

Invariant 7.4.26

For each point $v \in V$ and point $x \in V_t$ where (x, t) originates from v,

$$\forall i \in [k] : \ 0 \leq v_i - x_i \leq \sum_{j=1}^{t} w^j.$$

Proof. We prove the statement by induction on t; the base case $t = 0$ is trivial. For iteration t, for each v, x where (x, t) originates from v, we have $x_i - W < (h_j(x))_i \leq x_i$ for all $i \in [k]$, $j \in [s]$ by definition of h_j. By induction, $0 \leq v_i - x_i \leq \sum_{j=1}^{t-1} w^j$ for all $i \in [k]$. Therefore, the points $h_j(x) \in V_{t+1}$, which also originate from v, satisfy

$$v_i - (h_j(x))_i \geq v_i - x_i \geq 0 \qquad \text{and} \qquad v_i - (h_j(x))_i \leq v_i - x_i + W \leq \sum_{j=1}^{t-1} w^j + W = \sum_{j=1}^{t} w^j$$

for all $i \in [k]$, completing the induction. $\square$

Lemma 7.4.27

For each point $v \in V$ and iteration $t \in [T+1]$, the expected number of vertices $(x,t) \in V(H)$ that originate from v is $O(s)$.

Proof. Fix an iteration $t \in [T+1]$. Let $r := \sum_{j=1}^{t-1} w^j \leq 2w^{t-1}$; by Invariant 7.4.26, all points $x \in V_t$ such that (x,t) originates from v are within the box $B := [v_1 - r, v_1] \times [v_2 - r, v_2] \times \cdots \times [v_k - r, v_k]$.

For each trial $j \in [s]$, consider the set $S := \{h_j(x) : x \in B\}$; note that every y in lines 17 and 18 for this trial satisfies $y \in S$. We claim that this set has expected size $O(1)$. To see why, observe that for each $i \in [k]$, the value $(h_j(x))_i$ over all $x \in B$ takes two distinct values with probability r/W and one value with probability $1 - r/W$, and these events are independent over all i. Moreover, if $k' \leq k$ of them take two distinct values, then $|S| \leq 2^{k'}$, and this happens with probability $\binom{k}{k'}(\frac{r}{W})^{k'}(1-\frac{r}{W})^{k-k'}$. Overall, the expected size of $|S|$ is at most

$$\begin{aligned}\sum_{k'=0}^{k} \binom{k}{k'} \left(\frac{r}{W}\right)^{k'} \left(1 - \frac{r}{W}\right)^{k-k'} \cdot 2^{k'} &= \left(\frac{r}{W} \cdot 2 + \left(1 - \frac{r}{W}\right)\right)^k = \left(1 + \frac{r}{W}\right)^k \\ &\leq \left(1 + \frac{2w^{t-1}}{w^t}\right)^k = \left(1 + \frac{2}{\lceil \log n \rceil}\right)^{O(\log n)} = O(1).\end{aligned}$$

Over all s independent trials, the sets S together capture all points y such that (y,t) originates from v. The expected number of such points (y,t) is therefore at most $O(s)$. □

Proof of Last Routing (Lemma 7.4.14).

Proof. We can follow the proof of Lemma 7.4.12 to obtain (7.4), where $W := w^{T+1}$ in this case. By Invariant 7.4.26, for each point $v \in V$ and point $x \in V_{T+1}$ where (x,t) originates from v, we have $\mathbb{1}v - x \leq kw \sum_{j=1}^{T} w^j = O(kw^{T+1})$. By Assumption 7.4.7, the vertices $v \in V$ are at most $n^c \leq w^T$ apart from each other in ℓ_1 distance. This means that the points $x \in V_{T+1}$ are at most $O(kw^{T+1})$ apart in ℓ_1 distance. Therefore, the routing on lines 24 and 25 has cost $C \leq \sum_x |b_{T+1}(x)| \cdot O(kw^{T+1})$. Combining this with (7.4) gives

$$kw\mathsf{cost}(R^*) \geq \sum_x |b_{T+1}(x)| \cdot kw^{T+1} \geq \Omega(C),$$

which means $C \leq O(kw) \cdot \mathsf{cost}(R^*_{T+1})$, as desired. □

Computing the Matrix R**.** First, we can modify Algorithm 9 to construct the graph H without changing the running time, since every edge added to H can be charged to one execution of line 17.

Now for any vector $b \in \mathbb{R}^V$ not necessarily satisfying $\mathbb{1} \cdot b = 0$, let R_b be the value of R

once Algorithm 9 is run on b. First, we will henceforth assume the following for simplicity:

Assumption 7.4.28

For each b, every (x, y) is updated at most once in $R_b(x, y)$ throughout Algorithm 9.

Intuitively, Assumption 7.4.28 is true with probability 1 because two different randomly shifted grids in Algorithm 9 align perfectly with probability 0. More specifically, the probability that $h_j(x) = h_{j'}(x')$ for two distinct x, j and x', j' (possibly not even at the same iteration) is 0.

Lemma 7.4.29

Assuming Assumption 7.4.28, we have that w.h.p., for each b and iteration t,

$$\frac{1}{4}kw^t \sum_{x: b_t(x)\neq 0} |b_t(x)| \leq \sum_{x,y:\, b_t(x)\neq 0} |R_b(x,y)| \cdot \mathbb{1}x - y \leq \frac{3}{4}kw^t \sum_{x: b_t(x)\neq 0} |b_t(x)| \tag{7.7}$$

Proof. Fix some $x \in V_t$, and fix a coordinate $i \in [k]$. For each trial $j \in [s]$, the difference $x_i - (h_j(x))_i$ is a uniformly random number in $[0, W)$ (where $W := w^t$ as before). Define random variable $X_j := (x_i - (h_j(x))_i)/W \in [0, 1]$, and $\mu := \sum_j \mathbb{E}[X_j] = s/2$. Applying Chernoff bounds on the variables X_j with $\epsilon := 1/4$, we obtain

$$\Pr\left[\left|\sum_{j=1}^{s} X_j - \mu\right| \geq \epsilon\mu\right] \leq \exp(-\epsilon^2\mu/3) \leq \exp\left(-\Omega(s)\right) = n^{-\omega(1)}.$$

Therefore, with probability $1 - n^{-\omega(1)}$,

$$\sum_{j=1}^{s}(x_i - (h_j(x))_i) = \sum_{j=1}^{s} W X_i \in \left[\frac{s}{4}W,\, \frac{3s}{4}W\right].$$

Summing over all $i \in [k]$, we obtain

$$\sum_{j=1}^{s} \mathbb{1}x - h_j(x) = \sum_{j=1}^{s}\sum_{i=1}^{k}(x_i - (h_j(x))_i) \in \left[\frac{1}{4}ksW,\, \frac{3}{4}ksW\right].$$

At this point, let us assume that every statement holds in the proof so far, which is true w.h.p. Fix a demand vector b; by Assumption 7.4.28, each term in the sum $\sum_{x,y:\, b_t(x)\neq 0} |R_b(x,y)| \cdot \mathbb{1}x - y$ appears exactly once in line 17, so it must appear on iteration t. In particular, the terms can be exactly partitioned by x. Every x with $b_t(x) \neq 0$ contributes $\sum_{j=1}^{s} |b_t(x)/s| \cdot \mathbb{1}x - h_j(x)$ to the sum (line 17), which is within $\left[\frac{1}{4}kW|b_t(x)|, \frac{3}{4}kW|b_t(x)|\right]$. Summing over all x proves (7.7). □

Therefore, by Lemma 7.4.29, to estimate the final routing cost $\sum_{x,y} |R_b(x,y)| \cdot \mathbb{1}x - y$ by an $O(1)$ factor, it suffices to compute the value

$$\sum_t kw^t \sum_{x:b_t(x)\neq 0} |b_t(x)|. \tag{7.8}$$

Remark 7.4.30

The purpose of reducing to summing over the values $|b_t(x)|$ is to save a factor s in the running time; if we did not care about extra polylog(n) factors, we could do without it.

Assuming Assumption 7.4.28, our goal is to construct a sparse matrix R so that $\mathbb{1}Rb$ equals (7.8). To do so, our goal is to have each coordinate in Rb represent $kw^t b_t(x)$ for some t, x with $b_t(x) \neq 0$. This has the benefit of generalizing to general demands b by the following *linearity* property:

Claim 7.4.31

Every value $b_t(x)$ for $t \in \{0, 1, 2, \dots, T+1\}$, $x \in \mathbb{R}^k$ is a linear function in the entries of $b \in \mathbb{R}^V$.

Proof. We show this by induction on t; the base case $t = 0$ is trivial. For each $t > 0$, the initialization b_{t+1} as the zero function is linear in b, and by line 19, each update of some $b_{t+1}(y)$ adds a scalar multiple of some $b_t(x)$ to $b_{t+1}(y)$. Since $b_{t+1}(y)$ was linear in b before the operation, and since $b_t(x)$ is linear in b by induction, $b_{t+1}(y)$ remains linear in b. □

To exploit linearity, we consider the set of "basis" functions R_b where $b = \chi_v$ for some $v \in V$. (Again, note that χ_v is not a demand vector, but we do not require that property here.)

Proof of Lemma 7.4.16. We first show by induction on t that if $b_t(x) \neq 0$ for $x \in \mathbb{R}^k$, then (x, t) originates from v; the base case $t = 0$ is trivial. For each $t > 0$, the only way some $b_{t+1}(y)$ is updated (line 19) is if there exist $x \in \mathbb{R}^k$ with $b_t(x) \neq 0$ and $y = h_j(x)$ for some $j \in [s]$. By induction, x originates from v, and by definition of the history graph H, there is a directed edge $((x, t), (y, t+1))$ in H added when line 17 is executed for this pair x, y. Therefore, there is a path from $(v, 0)$ to $(y, 1)$ in H, and y also originates from v.

Therefore, for each t, the number of points $x \in \mathbb{R}^k$ satisfying $b_t(x)$ is at most the number of vertices $(x, t) \in V(H)$ originating from v, which by Lemma 7.4.27 is $O(s)$ in expectation.

Finally, the functions b_t can be computed by simply running Algorithm 9. $O(s)$ time is spent for each (x, t) with $b_t(x) \neq 0$ (assuming the entries of R_{χ_v} are stored in a hash table), giving $O(s^2)$ expected time for each iteration t. □

Proof of Lemma 7.4.17. We run Algorithm 9 for each demand χ_v *over the same randomness* (in particular, the same choices of h_j); define $b_t^{\chi_v}$ to be the function b_t on input χ_v. Let b_t

be the functions on input b. By linearity (Claim 7.4.31), we have that for each t, x,

$$b_t(x) = \sum_{v \in V} b(v) \cdot b_t^{\chi_v}(x). \tag{7.9}$$

By Lemma 7.4.16, the functions $b_t^{\chi_v}$ for all t, χ_v can be computed in $O(s^2 T n)$ total time in expectation.

We now construct matrix R as follows: for each t, x with $b_t^{\chi_v}(x) \neq 0$ for at least one χ_v, we add a row to R with value $kw^t b_t^{\chi_v}$ at each entry $v \in V$. The dot product of this row with b, which becomes a coordinate entry in Rb, is exactly

$$\sum_{v \in V} kw^t b_t^{\chi_v}(x) \cdot b(v) \overset{(7.9)}{=} kw^t b_t(x).$$

Hence, $\mathbb{1} Rb$ is exactly (7.8), which approximates the routing cost to factor $O(1)$ by Lemma 7.4.29, assuming Assumption 7.4.28 (which holds with probability 1). Finally, by Lemma 7.4.16, R has $O(sTn)$ entries in expectation.

Lastly, we address the issue that the algorithm only runs quickly in expectation, not w.h.p. Our solution is standard: run the algorithm $O(\log n)$ times, terminating it early each time if the running time exceeds twice the expectation. Over $O(\log n)$ tries, one will finish successfully w.h.p., so the final running time has an extra factor of $O(\log n)$, hence $O(s^2 T n \log n)$. $\square$

7.4.6 Parallel Transshipment

By inspection, the entire Algorithm 9 is parallelizable in $\tilde{O}(m)$ work and polylog(n) time. The only obstacle to the entire ℓ_1-oblivious routing algorithm is the initial ℓ_1-embedding step, and the only hurdle to the final proof of Theorem 7.1.4 is the final low-stretch spanning step. The latter we can handle with Remark 7.4.4, since minimum spanning tree can be computed in parallel with Boruvka's algorithm. We state the following corollary below to be used in our parallel algorithms.

7.5 Vertex Sparsification and Recursion

This section is dedicated to proving the vertex-sparsification lemma, Lemma 7.3.15, restated below:

Lemma: Restatement of Lemma 7.3.15

Let $G = (V, E)$ be a graph with n vertices and m edges, and let $\mathcal{A}$ be an algorithm that inputs any vertex set $S \subseteq V$ and outputs a $(1 + 1/\log n)$-approximate S-SSSP potential of G. Then, there is an algorithm that computes an ℓ_1-embedding of G into $O(\log^2 n)$ dimensions with distortion $O(\log^3 n)$ and calls $\mathcal{A}$ at most $O(\log^2 n)$ times, plus $\tilde{O}(m)$ additional work and polylog(n) additional time.

7.5.1 Case $S = \{s\}$ of Lemma 7.3.15

In this section, we first prove Lemma 7.3.15 for the case when $S = \{s\}$ for a single source $s \in V$ in the lemma below. We then extend our result to any set $S \subseteq V$ in Section 7.5.4.

Lemma 7.5.2: s-SSSP algorithm

Let $G = (V, E)$ be a connected graph with n vertices and $(n - 1) + t$ edges, and let $\alpha > 0$ be a parameter. Let $\mathcal{A}$ be an algorithm that inputs a connected graph on at most $5t$ vertices and edges and outputs an α-approximate s-SSSP potential of that graph. Then, for any source $s \in V$, we can compute an α-approximate s-SSSP potential of G through a single call to $\mathcal{A}$, plus $\tilde{O}(m)$ additional work and polylog(n) additional time.

Our approach is reminiscent of the j-tree construction of Madry [77], but modified to handle SSSP instead of flow/cut problems.[5]

First, compute a spanning tree T of G, and let $S_0 \subseteq V$ be the endpoints of the t edges in $G - T$ together with the vertex s, so that $|S_0| \leq 2t + 1$. Next, let T_0 be the (tree) subgraph in T whose edges consist of the union of all paths in T between some pair of vertices in S_0. The set T_0 can be computed in parallel as follows:

1. Root the tree T arbitrarily, and for each vertex $v \in V$, compute the number $N(v)$ of vertices in S_0 in the subtree rooted at v.
2. Compute the vertex v with maximum depth satisfying $N(v) = |S_0|$; this is the lowest common ancestor $\mathsf{lca}(S_0)$ of the vertices in S_0.
3. The vertices in T_0 are precisely the vertices v in the subtree rooted at $\mathsf{lca}(S_0)$ which satisfy $N(v) \neq 0$.

Let S_3 be the set of vertices in T_0 whose degree in T_0 is at least 3, and let $S := S_0 \cup S_3$. Starting from T_0, contract every maximal path of degree-2 vertices disjoint from S into a single edge whose weight is the sum of weights of edges on that path; let T_1 be the resulting tree. Since every leaf in T_0 is a vertex in S_0, and since every degree-2 vertex disjoint from S is contracted, the vertex set of T_1 is exactly S. We furthermore claim the following:

[5]Essentially, according to the terminology of [77], any graph with $(n - 1) + t$ edges is a $5t$-tree

Claim 7.5.3

T_1 has at most $4t$ vertices and edges.

Proof. Let n_1, n_2, and $n_{\geq 3}$ be the number of vertices in T_1 of degree 1, 2, and at least 3, respectively. Since every leaf in T_0 is a vertex in S_0, we have $n_1 \geq |S_0|$. Also, since T_1 is a tree, it has $n_1 + n_2 + n_{\geq 3} - 1$ edges, and since the sum of degrees is twice the number of edges, we have

$$n_1 + 2n_2 + 3n_{\geq 3} \leq 2(n_1 + n_2 + n_{\geq 3} - 1) \implies n_{\geq 3} \leq n_1 - 2 \leq |S_0| - 2.$$

The number of vertices in T_1 is exactly $n_1 + n_{\geq 3}$, which is at most $2|S_0| - 2 \leq 4t$. The edge bound also follows since T_1 is a tree. □

Let G_1 be T_1 together with each edge in $G - T$ added to its same endpoints (recall that no endpoint in $G - T$ is contracted). Since T_1 has at most $4t$ vertices and edges by Claim 7.5.3, and since we add t additional edges to form G_1, the graph G_1 has at most $4t$ vertices and $5t$ edges.

Finally, let G_0 be T_0 together with each edge in $G - T$ added to its same endpoints, so that G_0 is exactly G_1 with the contracted edges expanded into their original paths. Since every edge in $G - T$ is contained in G_0, we have that $G - G_0$ is a forest. We summarize our graph construction below, which will be useful in Section 7.5.4.

Lemma 7.5.4

Let $G = (V, E)$ be a graph with n vertices and $(n - 1) + t$ edges, and let T be an arbitrary spanning tree of G. We can select a vertex set $V_0 \subseteq V$ and define the graph $G_0 := G[V_0]$ such that (i) $G - G_0$ is a forest, and (ii) we can contract degree-2 paths from G_0 into single edges so that the resulting graph G_1 has at most $4t$ vertices and $5t$ edges. The contracted edges in G_1 have weight equal to the total weight of the contracted path. This process takes $\tilde{O}(m)$ work and $\text{polylog}(n)$ time.

It is easy to see that the aspect ratio of G_1 is $O(M)$. Now, call $\mathcal{A}$ on G_1 with s as the source (recall that $s \in S_0 \subseteq S = V(G_1)$, so it is a vertex in G_1), obtaining an SSSP potential ϕ_1 for G_1. It remains to extend ϕ_1 to the entire vertex set V.

7.5.2 Extending to Contracted Paths

First, we extend ϕ_1 to the vertices (of degree 2) contracted from T_0 to T_1. More precisely, we will compute a SSSP potential $\phi_0(v)$ on the vertices in G_0 that agrees with ϕ_1 on $V(G_1)$.

Define $\phi_0(v) := \phi_1(v)$ for $v \in V(G_1)$, and for each such path $v_0, v_1, \ldots, v_\ell$ with $v_0, v_\ell \in$

$V(G_1)$ we extend ϕ_0 to $v_1, \ldots, v_{\ell-1}$ as follows:

$$\phi_0(v_j) := \min \left\{ \phi_1(v_0) + \sum_{i=1}^{j} w(v_{i-1}, v_i),\ \phi_1(v_\ell) + \sum_{i=j}^{\ell-1} w(v_i, v_{i+1}) \right\};$$

note that these values are the same if we had replaced the path by its reverse $(v_\ell, v_{\ell-1}, \ldots, v_0)$ instead.

Claim 7.5.5

For all $u, v \in V(G_1)$, we have $d_{G_0}(u,v) = d_{G_1}(u,v)$.

Proof. Observe that any simple path P in G_0 between $u, v \in V(G_1)$ must travel entirely along any path of degree-2 vertices sharing an edge with P. Therefore, for every contracted path in G that shares an edge with P, we can imagine contracting that path inside P as well. Since paths of degree-2 are contracted to an edge whose weight is the sum of weights of edges along that path, the total weight of P does not change. Since P is now a path in G_1, this shows that $d_{G_1}(u,v) \le d_{G_0}(u,v)$. Conversely, any path in G_1 can be "un-contracted" into a path in G_0 of the same length, so we have $d_{G_0}(u,v) \le d_{G_1}(u,v)$ as well, and equality holds. □

Claim 7.5.6

The vector ϕ_0 is an α-approximate s-SSSP potential of G_0.

Proof. We first prove property (2). Since $\phi_0(v) = \phi_1(v)$ for $v \in V(G_1)$, property (2) holds for ϕ_0 for edges G_1 that were not contracted from a path in G_0. For an edge (u,v) that was contracted, there is a contracted path $v_0, v_1, \ldots, v_\ell$ where $u = v_j$ and $v = v_{j+1}$ for some j. First, suppose that

$$\phi_0(v_j) = \phi_1(v_0) + \sum_{i=1}^{j} w(v_{i-1}, v_i) \iff \phi_1(v_0) + \sum_{i=1}^{j} w(v_{i-1}, v_i) \le \phi_1(v_\ell) + \sum_{i=j}^{\ell-1} w(v_i, v_{i+1}).$$

Then,

$$\phi_0(v_{j+1}) - \phi_0(v_j) \le \left(\phi_1(v_0) + \sum_{i=1}^{j+1} w(v_{i-1}, v_i) \right) - \left(\phi_1(v_0) + \sum_{i=1}^{j} w(v_{i-1}, v_i) \right) = w(v_j, v_{j+1}).$$

Otherwise, if

$$\phi_0(v_j) = \phi_1(v_\ell) + \sum_{i=j}^{\ell-1} w(v_i, v_{i+1}) \iff \phi_1(v_0) + \sum_{i=1}^{j} w(v_{i-1}, v_i) \ge \phi_1(v_\ell) + \sum_{i=j}^{\ell-1} w(v_i, v_{i+1}),$$

then

$$\phi_0(v_{j+1}) \le \phi_1(v_\ell) + \sum_{i=j+1}^{\ell-1} w(v_i, v_{i+1}) = \phi_1(v_\ell) + \sum_{i=j}^{\ell-1} w(v_i, v_{i+1}) - w(v_j, v_{j+1}) \le 0.$$

Therefore, in both cases, $\phi_0(v) - \phi_0(u) = \phi_0(v_{j+1}) - \phi_0(v_j) \le w(v_j, v_{j+1})$. For the other direction $\phi_0(v) - \phi_0(u) \le w(v_j, v_{j+1})$, we can simply swap u and v.

We now focus on property (1). Since $\phi_0(v) = \phi_1(v)$ for $v \in V(G_1)$, and since $d_{G_0}(u, v) = d_{G_1}(u, v)$ for $u, v \in V(G_1)$ by Claim 7.5.5, property (1) holds for $u, v \in V(G_1)$. We now prove property (1) for vertices $v \notin V(G_1)$.

If $v \notin V(G_1)$, then $v = v_j$ for some path $v_0, v_1, \ldots, v_\ell$ contracted in G_0 ($v_0, v_\ell \in V(G_1)$). Observe that $d_0 := d_{G_0}(s, v_0) + \sum_{i=1}^{j} w(v_{i-1}, v_i)$ is the shortest length of any (simple) path from s to v that passes through v_0, and similarly, $d_\ell := d_{G_0}(s, v_\ell) + \sum_{i=j+1}^{\ell} w(v_{i-1}, v_i)$ is the shortest length of any (simple) path from s to v that passes through v_ℓ. Furthermore, $d_{G_0}(s, v) = \min\{d_0, d_\ell\}$. We have

$$\begin{aligned}\left(\phi_1(v_0) + \sum_{i=1}^{j} w(v_{i-1}, v_i)\right) - \phi_1(s) &\ge \sum_{i=1}^{j} w(v_{i-1}, v_i) + \frac{1}{\alpha} \cdot d_{G_0}(s, v_0) \\ &\ge \frac{1}{\alpha}\left(\sum_{i=1}^{j} w(v_{i-1}, v_i) + d_{G_0}(s, v_0)\right) = \frac{d_0}{\alpha},\end{aligned}$$

and similarly,

$$\begin{aligned}\left(\phi_1(v_\ell) + \sum_{i=j}^{\ell-1} w(v_i, v_{i+1})\right) - \phi(s) &\ge \sum_{i=j}^{\ell-1} w(v_i, v_{i+1}) + \frac{1}{\alpha} \cdot d_{G_0}(s, v_\ell) \\ &\ge \frac{1}{\alpha}\left(\sum_{i=j}^{\ell-1} w(v_i, v_{i+1}) + d_{G_0}(s, v_\ell)\right) = \frac{d_\ell}{\alpha}.\end{aligned}$$

It follows that

$$\begin{aligned}\phi_0(v_j) - \phi_0(s) &= \min\left\{\phi_1(v_0) + \sum_{i=1}^{j} w(v_{i-1}, v_i),\ \phi_1(v_\ell) + \sum_{i=j}^{\ell-1} w(v_i, v_{i+1})\right\} \\ &\ge \min\left\{\frac{d_0}{\alpha},\ \frac{d_\ell}{\alpha}\right\} \\ &= \frac{1}{\alpha} \cdot d_{G_0}(s, v),\end{aligned}$$

proving property (1). □

7.5.3 Extending to Forest Components

It remains to extend ϕ_0 to an SSSP potential in the original graph G. First, recall that all edges in $G-T$ have endpoints inside $S = V(G_1) \subseteq V(G_0)$, which means that $G - E(G_0)$ is a forest contained in T. Moreover, since $G_0 \cap T = T_0$ is connected, every connected component (tree) in $G - E(G_0)$ shares exactly one endpoint with $V(G_0)$ (otherwise, there would be a cycle in T). Therefore, any simple path between two vertices in $V(G_0)$ must be contained in G_0. Since G_0 is itself an induced subgraph of G, in particular with the same edge weights, we have $d_G(u,v) = d_{G_0}(u,v)$ for all $u, v \in V(G_0)$.

In addition, for each component (tree) C in the forest $G - E(G_0)$ that shares vertex r with $V(G_0)$ (which could possibly be s), any path from s to a vertex in C must pass through r. In particular, the shortest path from s to a vertex $v \in C$ consists of the shortest path from s to r (possibly the empty path, if $r = s$) concatenated with the (unique) path in C from r to v. It follows that $d_G(s,v) = d_G(s,r) + d_C(r,v)$.

With these properties of G in mind, let us extend ϕ_0 to the potential ϕ on V as follows: for $v \in V(G_0)$, define $\phi(v) := \phi_0(v)$, and for each connected component C of $G - E(G_0)$ sharing vertex r with $V(G_0)$, define $\phi(v) = \phi_0(r) + d_C(r,v)$. Since C is a tree, the values $d_C(r,v)$ for each $v \in V(C)$ are easily computed in parallel.

Claim 7.5.7

The vector ϕ is an α-approximate s-SSSP potential of G.

Proof. Since ϕ_0 and ϕ agree on $V(G_0)$, and since $G[V(G_0)]$ and G_0 agree on their edges (including their weights), property (1) of Definition 7.2.7 holds for all $v \in V(G_0)$ and property (2) holds for all $u, v \in V(G_0)$.

Now fix a connected component C of $G - E(G_0)$ sharing vertex r with $V(G_0)$. For each vertex $v \in C$, we have

$$\begin{aligned}\phi(v) - \phi(s) = \big(\phi_0(r) + d_C(r,v)\big) - \phi_0(s) \geq \frac{1}{\alpha} \cdot d_G(s,r) + d_C(r,v) &\geq \frac{1}{\alpha}\big(d_G(s,r) + d_C(r,v)\big) \\ &= \frac{d_G(s,v)}{\alpha},\end{aligned}$$

proving property (1) for vertices in C. For property (2), consider an edge (u,v) in C. Since C is a tree, either $d_C(r,u) = d_C(r,v) + w(u,v)$ or $d_C(r,v) = d_C(r,u) + w(u,v)$, so in both cases,

$$|\phi(u) - \phi(v)| = \big|\big(\phi_0(r) + d_C(r,u)\big) - \big(\phi_0(r) + d_C(r,v)\big)\big| = \big|d_C(r,u) - d_C(r,v)\big| = w(u,v),$$

proving property (2). □

7.5.4 Generalizing to S-SSSP

Of course, Lemma 7.3.15 requires calls to not just s-SSSP, but S-SSSP for a vertex subset $S \subseteq V$. In this section, we generalize the algorithm to work for S-SSSP for any $S \subseteq V$.

Lemma 7.5.8: S-SSSP algorithm

Let $G = (V, E)$ be a connected graph with n vertices and $(n-1)+t$ edges, and let $\alpha > 0$ be a parameter. Let $\mathcal{A}$ be an algorithm that inputs (i) a connected graph on at most $70t$ vertices and edges with aspect ratio $O(M)$ and (ii) a source vertex s, and outputs an α-approximate s-SSSP potential of that graph. Then, for any subset $S \subseteq V$, we can compute an α-approximate S-SSSP potential of G through a single call to $\mathcal{A}$, plus $\tilde{O}(m)$ additional work and $\text{polylog}(n)$ additional time.

Let G_0 and G_1 be the graphs guaranteed from Lemma 7.5.4, and let $V_0, V_1 \subseteq V$ be their respective vertex sets. Consider the set $\mathcal{C}$ of connected components (trees) in $G - G_0$; for each component $C \in \mathcal{C}$, let $r(C)$ be the vertex shared between C and V_0, and let $\mathcal{C}$ be the set of components C with $S \cap (V(C) \setminus \{r(C)\}) \neq \emptyset$. Root each component $C \in \mathcal{C}$ at $r(C)$, and let $C^\uparrow$ be the subgraph of C induced by the vertices that have no ancestor in $S \cap (V(C) \setminus \{r(C)\})$ (see Figure 7.5.4); we will first focus our attention on $C^\uparrow$. Let $d(C) := d_C(r(C), S \cap V(C)) = d_{C^\uparrow}(r(C), S \cap V(C^\uparrow))$ be the distance from $r(C)$ to the closest vertex in $S \cap V(C)$, which must also be in $S \cap V(C^\uparrow)$.

Next, consider all paths P in G_0 that were contracted into edges in C_1; for each such path P, let $r_1(P), r_2(P) \in V_1$ be the two endpoints of P. Let $\mathcal{P}$ be the paths P which satisfy $S \cap (V(P) \setminus \{r_1(P), r_2(P)\}) \neq \emptyset$. For $i = 1, 2$, let $v_i(P)$ be the vertex on P closest to $r_i(P)$, and define $d_i(P) := d_P(r_i(P), v_i(P)) = d_P(r_i(P), S \cap V(P))$. Note that it is possible that $v_1(P) = v_2(P)$, which happens precisely when $|S \cap V(P)| = 1$. Define $P^\uparrow$ to be the union of the path from $r_1(P)$ to $v_1(P)$ and the path from $r_2(P)$ to $v_2(P)$. Again, we first focus on $P^\uparrow$.

We construct a graph G_2 as follows. The vertex set is $V_2 := V_1 \cup \bigcup_{C \in \mathcal{C}} V(C^\uparrow) \cup \bigcup_{P \in \mathcal{P}} V(P^\uparrow) \cup \{s\}$ for a new vertex s. Add the graph G_1 onto the vertices V_1, and for each $C \in \mathcal{C}$ and $P \in \mathcal{P}$, add the graphs $C^\uparrow$ and $P^\uparrow$ into $V(C^\uparrow)$ and $V(P^\uparrow)$, respectively. For each vertex $v \in S \cap V_1$, add an edge of weight 0 between s and v, adding a total of $|S \cap V_1| \leq |V_1| \leq 4t$ edges. Next, for each $C \in \mathcal{C}$, add an edge from s to $r(C)$ of weight $d(C)$, and for each $P \in \mathcal{P}$, add an edge from s to $r_i(P)$ of weight $d_i(P)$ for $i = 1, 2$. Since every component $C \in \mathcal{C}$ has a distinct $r(C) \in V_1$, we have $|\mathcal{C}| \leq |V_1| \leq 4t$. Since every path $P \in \mathcal{P}$ gets contracted to a (distinct) edge in G_1, we have $|\mathcal{P}| \leq |E(G_1)| \leq 5t$. Therefore, we add at most $13t$ edges from s.

Claim 7.5.9

G_2 has at most $(|V_2| - 1) + 14t$ edges, and for every vertex $v \in V_2 \setminus \{s\}$, we have $d_{G_2}(s, v) \geq d_G(S, v)$.

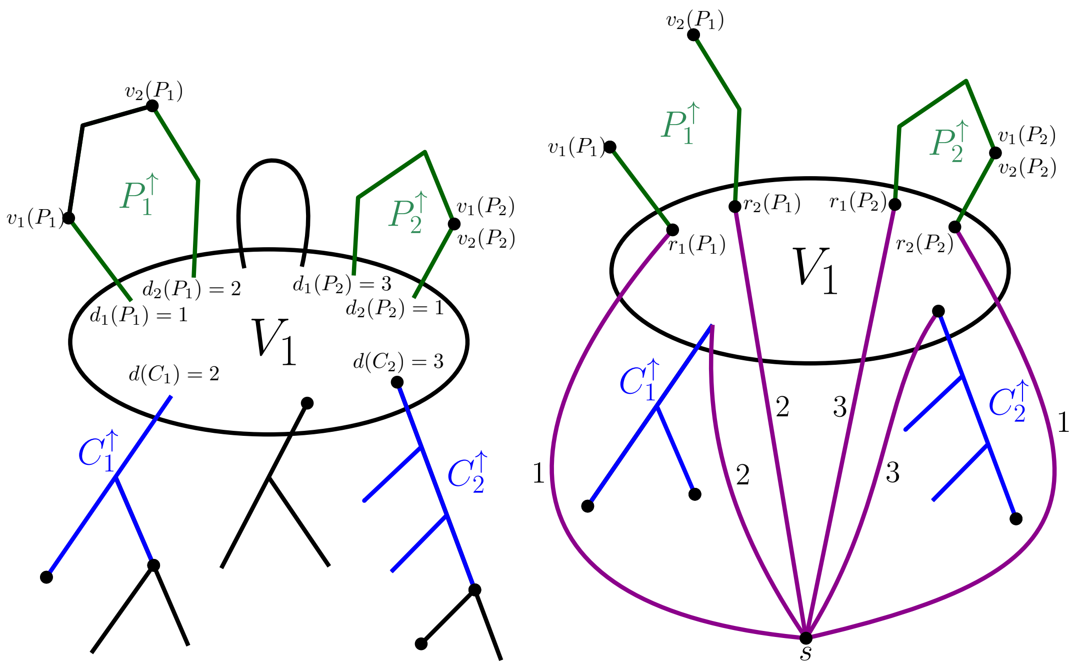

Figure 7.3: Construction of the graph G_2.

Proof. Since G has $(n-1)+t$ edges, there exists some t edges $F \subseteq E$ such that $G - F$ is a tree. This means that $G[V_2 \setminus \{s\}]$ has at most $(|V_2 \setminus \{s\}| - 1)$ edges in $G - F$, and since $|F| = t$ and we added at most $13t$ extra edges, G_2 has at most $(|V_2 \setminus \{s\}| - 1) + 14t$ edges.

To prove the second statement, consider a vertex $v \in V_2 \setminus \{s\}$, and let P be the shortest path from s to v in G_2. If the first edge on the path (adjacent to s) its other endpoint (besides s) inside $V_1 \cap S$, then this edge has weight 0, and the path P minus that first edge is a path in G_1 from S to v of equal weight. Then, for each edge on the path formed by contracting a path in G to an edge in G_1, we can expand the edge back to the contracted path, obtaining a path the same weight in G.

Next, suppose that the first edge connects to vertex $r(C)$ for some $C \in \mathcal{C}$. In this case, we replace that edge with the path in $C^\uparrow$ from $S \cap V(C^\uparrow)$ to $r(C^\uparrow)$ of weight $d(C)$. The new path is a path in G_1 from V to v of the same weight, and we can expand contracted edges as in the first case.

Otherwise, the first edge must connect to a vertex $r_i(P)$ for some $P \in \mathcal{P}$ and $i \in \{1, 2\}$. In this case, we similarly replace that edge with the path in $P^\uparrow$ from $S \cap T(P^\uparrow)$ to $r_i(P^\uparrow)$ of weight $d(P)$, and the rest of the argument is analogous. $\square$

It is clear that G_2 has aspect ratio $O(M)$. We now apply Lemma 7.5.2 on G_2 with source s and algorithm $\mathcal{A}$ (note that $70t = 5 \cdot 14t$), obtaining an s-SSSP potential ϕ_2 on V_2. W.l.o.g., we can assume that $\phi_2(s) = 0$, since we can safely add any multiple of $\mathbb{1}$ to $\phi_2(s)$. We now extend ϕ_2 to V by setting $\phi_2(v) := \infty$ for all $v \in V \setminus V_2$.

We next define a potential $\phi_{\mathcal{C}}$ on V as follows. For each $C \in \mathcal{C}$, let $\phi_{\mathcal{C}}(r(C)) := \infty$, let

$\phi_{\mathcal{C}}(v) := d_C(S \cap (V(C) \setminus \{r(C)\}), v)$ for $v \in V(C) \setminus \{r(C)\}$ (that is, *exact* distances in C from $S \cap (V(C) \setminus \{r(C)\})$, and assume w.l.o.g. that $\phi_{\mathcal{C}}(v) = 0$ for all $v \in S \cap V(C)$ (see Observation 7.2.10). For all remaining $v \in V \setminus \bigcup_{C \in \mathcal{C}} V(C)$, define $\phi_{\mathcal{C}}(v) := \infty$. Similarly, we define potential $\phi_{\mathcal{P}}$ as follows. For each $P \in \mathcal{P}$, let $\phi_{\mathcal{P}}(r_1(P)) = \phi_{\mathcal{P}}(r_2(P)) := \infty$, let $\phi_{\mathcal{P}}(v) := d_P(S \cap (V(P) \setminus \{r_1(P), r_2(P)\}), v)$ for $v \in V(P)$, and assume w.l.o.g. that $\phi_{\mathcal{P}}(v) = 0$ for all $v \in S \cap V(P)$; for all remaining $v \in V \setminus \bigcup_{P \in \mathcal{P}} V(P)$, define $\phi_{\mathcal{P}}(v) := \infty$. Since each $C \in \mathcal{C}$ and $P \in \mathcal{P}$ is a tree, this can be done efficiently in parallel as stated below, whose proof we defer to Section 7.10.3.

Lemma 7.5.10

Given a tree $T = (V, E)$ and a set of sources $S \subseteq V$, we can compute an exact S-SSSP potential in $\tilde{O}(m)$ work and polylog(n) time.

Finally, define $\phi(v) := \min\{\phi_2(v), \phi_{\mathcal{C}}(v), \phi_{\mathcal{P}}(v)\}$. Note that for all $v \in V$, we have $\phi_i(v) \geq 0$ for all $i \in \{2, \mathcal{C}, \mathcal{P}\}$ by Observation 7.2.10, so $\phi(v) \geq 0$ as well.

Claim 7.5.11

The vector ϕ is an α-approximate S-SSSP potential of G.

Proof. Since $\phi(s) = 0$ for all $s \in S$, property (0) of Definition 7.2.12 holds. We now prove property (1). Fix a vertex $v \in V$, and suppose first that $\phi(v) = \phi_2(v)$ (i.e., the minimum is achieved at $\phi_2(v)$). Then, by Claim 7.5.9, we have $d_{G_2}(s, v) \geq d_G(s, v)$. This, along with the guarantee $\phi_2(v) \geq \frac{1}{\alpha} d_{G_2}(s, v)$ from ϕ_2, implies that $\phi_2(v) \geq \frac{1}{\alpha} d_{G_2}(s, v) \geq \frac{1}{\alpha} d_G(s, v)$. Now suppose that $\phi(v) = \phi_{\mathcal{C}}(v)$. Since $\phi_{\mathcal{C}}(v) = d_C(v, S \cap V(C))$ for some $C \in \mathcal{C}$, we have

$$\phi(v) = d_C(v, S \cap V(C)) \geq d_G(v, S \cap V(C)) \geq d_G(v, S) \geq \frac{1}{\alpha} d_G(v, S).$$

The remaining case $\phi(v) = \phi_{\mathcal{P}}(v)$ is analogous, with every instance of C and $\mathcal{C}$ replaced by P and $\mathcal{P}$, respectively.

We now focus on property (2). Note that if $\phi_2, \phi_{\mathcal{C}}, \phi_{\mathcal{P}}$ each satisfied property (2), then by Observation 7.2.11, ϕ would as well. In fact, a more fine-grained variant of Observation 7.2.11 states that for any edge $(u, v) \in E$, if we have $|\phi_i(u) - \phi_i(v)| \leq d_G(u, v)$ for each $i \in \{2, \mathcal{C}, \mathcal{P}\}$, then $|\phi(u) - \phi(v)| \leq d_G(u, v)$ as well. Hence, we only need to consider edges for which $|\phi_i(u) - \phi_i(v)| > d_G(u, v)$ for some $i \in \{2, \mathcal{C}, \mathcal{P}\}$.

We first focus on $i = 2$. Observe that by property (1) on ϕ_2, the only edges $(u, v) \in E$ for which $|\phi_2(u) - \phi_2(v)| > d_G(u, v)$ are those where $\phi_2(u) < \infty$ and $\phi_2(v) = \infty$ or vice versa.[6] Let us assume w.l.o.g. that $\phi_2(u) < \infty$ and $\phi_2(v) = \infty$; by construction, we must either have $u \in S \cap V(C^{\uparrow})$ for some $C \in \mathcal{C}$ or $u \in S \cap V(P^{\uparrow})$ for some $P \in \mathcal{P}$. In the former case, since $\phi_{\mathcal{C}}(u) = 0$ and $\phi(u) \geq 0$, we must have $\phi(u) = 0$, and since $\phi_{\mathcal{C}}(v) < \infty$ and

[6]Let us assume for simplicity that $\infty - \infty = 0$. To be more formal, we should replace each ∞ with some large number M that exceeds the weighted diameter of the graph, so that we have $M - M = 0$ instead.

$\phi_2(v) = \phi_{\mathcal{P}}(v) = \infty$, we also have $\phi(v) = \phi_{\mathcal{C}}(v)$. We thus have, for some $C \in \mathcal{C}$,

$$|\phi(u) - \phi(v)| = |\phi_{\mathcal{C}}(u) - \phi_{\mathcal{C}}(v)| = |0 - \phi_{\mathcal{C}}(v)| = \phi_{\mathcal{C}}(v) = d_C(v, S \cap V(C)) \leq d_G(u, v),$$

so edge (u, v) satisfies property (2), as needed.

Next, consider the case $i = \mathcal{C}$. By construction, the only edges $(u, r) \in E$ for which $|\phi_{\mathcal{C}}(u) - \phi_{\mathcal{C}}(r)| > d_G(u, v)$ are those where $\phi_{\mathcal{C}}(u) < \infty$ and $\phi_{\mathcal{C}}(r) = \infty$ or vice versa. Assuming again that $\phi_{\mathcal{C}}(u) < \infty$ and $\phi_{\mathcal{C}}(r) = \infty$, we must have $u \in V(C)$ and $r = r(C)$ for some $C \in \mathcal{C}$. By construction, we must have $\phi(r) = \phi_2(r)$ and $\phi(u) = \min\{\phi_2(u), \phi_{\mathcal{C}}(u)\}$. By property (2) of ϕ_2, we have $|\phi_2(u) - \phi_2(r)| \leq w(u, r)$, so it suffices to show that $\phi_{\mathcal{C}}(u) \geq \phi(r) - w(u, r)$, from which $|\phi(u) - \phi(r)| \leq w(u, r)$ will follow.

By Observation 7.2.9 and the fact that $\phi(s) = 0$, we have $\phi_2(r) \leq d_{G_2}(s, r)$. Also, by construction of G_2, we have $d_{G_2}(s, r) \leq w_{G_2}(s, r) = d_C(S \cap (V(C) \setminus \{r(C)\})$. Therefore,

$$\begin{aligned}
\phi_{\mathcal{C}}(u) = d_C(S \cap (V(C) \setminus \{r(C)\}), u) &= d_C(S \cap (V(C) \setminus \{r(C)\}), r) - w(u, r) \\
&\geq d_{G_2}(u, r) - w(u, r) \\
&\geq \phi_2(r) - w(u, r) \\
&= \phi(r) - w(u, r),
\end{aligned}$$

as desired.

The case $i = \mathcal{P}$ is almost identical to the case $i = \mathcal{C}$, except we now have $r = r_1(P)$ or $r = r_2(P)$. Since the rest of the argument is identical, we omit the proof. □

7.6 Ultra-spanner Algorithm

In this section, we present our algorithm for constructing an ultra-spanner. It is a modification of the weighted spanner algorithm of [82], where we sacrifice more factors in the spanner approximation for the needed ultra-sparsity.

Lemma: Restatement of Lemma 7.3.14

Given a weighted graph G with polynomial aspect ratio and a parameter $k \geq \Omega(1)$, there is an algorithm to compute a k^2-spanner of G with $(n-1) + O(\frac{m \log n}{k})$ edges in $\tilde{O}(m)$ work and polylog(n) time.

Our ultra-spanner algorithm closely resembles the weighted spanner algorithm of [82]. Their algorithm outputs an $O(k)$-spanner with $O(n^{1+1/k} \log k)$ edges, which is not ultra-sparse and therefore insufficient for our purposes. However, the $\log k$ factor of their algorithm comes from splitting the graph into $O(\log k)$ separate ones, computing a spanner for each, and taking the union of all spanners. We modify their algorithm to consider only one graph, at the cost of an extra k-factor in the stretch, which is okay for our application. We first introduce the subroutine `ESTCluster` for *unweighted* graphs from [82] (which dates back to [83]) and its guarantee, whose proof we sketch for completeness. Note that while this algorithm can be adapted to the weighted setting, executing the algorithm efficiently in parallel is difficult

Algorithm 10 `ESTCluster`$(G=(V,E),\beta\in(0,1])$, G is *unweighted*

```
1: For each vertex $u$, sample $\delta_u$ independently from the geometric distribution with mean $1/\beta$
2: Create clusters by defining $C_u := \{v \in V : u = \arg\min_{u' \in V} d(u',v) - \delta_{u'}\}$, with ties broken by a universal linear ordering of $V$. If $u \in C_u$, then $u$ is the center of cluster $C_u$
3: Return the clusters $C_u$ along with a spanning tree on each cluster rooted at its center.
```

Lemma 7.6.2

For each edge in E, the probability that its endpoints belong to different clusters is at most β.

Proof. Fix an edge $(v,v') \in E$, and let $u_1,u_2 \in V$ be the vertices achieving the smallest and second-smallest values of $d(u',v)-\delta_{u'}$ over all $u' \in V$, with ties broken by the linear ordering of V. (In particular, $v \in C_{u_1}$.) Let us condition on the choices of u_1,u_2 and the value of $d(u_2,v)-\delta_{u_2}$. First, suppose that $u_1 \le u_2$ in the linear ordering (that is, u_1 is preferred in the event of a tie). Then, we know that

$$v' \in C_{u_1} \iff d(u_1,v')-\delta_{u_1} \le d(u_2,v')-\delta_{u_2} \iff \delta_{u_1} \ge d(u_1,v')-d(u_2,v')+\delta_{u_2}.$$

So far, we are conditioning on the event $\delta_{u_1} \ge d(u_1,v)-d(u_2,v)+\delta_{u_2}$. By the memoryless property of geometric variables, with probability $1-\beta$, we have $\delta_{u_1} \ge (d(u_1,v)-d(u_2,v)+\delta_{u_2})+1$. In that case, we also have

$$d(u_1,v')-\delta_{u_1} \le (d(u_1,v)+1)-\delta_{u_1} \le (d(u_1,v)+1)-(d(u_1,v)-d(u_2,v)+\delta_{u_2}+1) = d(u_2,v)+\delta_{u_2},$$

so $v' \in C_{u_1}$ as well and edge (v,v') lies completely inside C_{u_1}.

If $u_1 \ge u_2$ in the linear ordering instead, then we know that

$$v \in C_{u_1} \iff d(u_1,v)-\delta_{u_1} < d(u_2,v)-\delta_{u_2} \iff \delta_{u_1} > d(u_1,v)-d(u_2,v)+\delta_{u_2},$$

and the proof proceeds similarly.

Overall, for each edge in E, the probability that its endpoints belong to different clusters is at most β. □

We now proceed to our ultra-spanner algorithm. Without loss of generality, the edge weights of G range from 1 to W for some $W=\operatorname{poly}(n)$. For positive real numbers x and k, define $\llbracket x \rrbracket_k := \max\{k^\alpha : \alpha \in \mathbb{Z},\ k^\alpha \le x\}$ as the largest integer power of k less than or equal to x. Let $\llbracket G \rrbracket_k = (V, \llbracket E \rrbracket_k)$ be the graph G with the weight $w(u,v)$ of each edge $(u,v) \in E$ replaced by $\llbracket w(u,v) \rrbracket_k$, so that in particular, all edge weights in G are now nonnegative integer powers of k. For each $\alpha \in \{0, 1, 2, \ldots, \llbracket W \rrbracket_k\}$, define $E_\alpha \subseteq \llbracket E \rrbracket_k$ as the set of edges in $\llbracket G \rrbracket_k$ with weight k^α.

Algorithm 11 `Ultraspanner`($\llbracket G \rrbracket_k = (V, \llbracket E \rrbracket_k)$)

```
1: Initialize $H_0 \leftarrow \emptyset$        ▷ $H_i \subseteq \llbracket E \rrbracket_k$ will be edges contracted over the iterations
2: Initialize $\llbracket S \rrbracket_k \leftarrow \emptyset$        ▷ $\llbracket S \rrbracket_k \subseteq \llbracket E \rrbracket_k$ will be the edges in the spanner
3: for $\alpha = 0, 1, 2, \ldots, \llbracket W \rrbracket_k$ do
4:     Let $V_0^{\alpha-1}, V_1^{\alpha-1}, \ldots$ be the connected components of $H_{\alpha-1}$
5:     Let $\Gamma_\alpha$ be the graph formed by starting with $(V, E_\alpha)$ and contracting each $V_i^{\alpha-1}$ into
       a single vertex
6:     Run ESTCluster on the (unweighted version of) $\Gamma_\alpha$ with $\beta := \frac{C \ln n}{k}$ on $\Gamma_\alpha$ for suffi-
       ciently large $C$, and let $F \subseteq \Gamma_\alpha$ be the forest returned
7:     Update $\llbracket S \rrbracket_k \leftarrow \llbracket S \rrbracket_k \cup F$
8:     Set $H_\alpha \leftarrow H_{\alpha-1} \cup F$
9:     Add to $\llbracket S \rrbracket_k$ all edges $e$ in $\Gamma_\alpha$ whose endpoints lie in different connected components
       of $F$
10: return $\llbracket S \rrbracket_k$ as the spanner
```

Lemma 7.6.3

If `Ultraspanner` *succeeds* (for a notion of success to be mentioned), then the output $\llbracket S \rrbracket_k$ is a k-spanner with at most $n - 1 + O(\frac{m \log n}{k})$ edges. We can define our success condition so that it happens with probability at least 1/3, and that we can detect if the algorithm fails (so that we can start over until it succeeds). Altogether, we can run the algorithm (repeatedly if necessary) so that w.h.p., the output $\llbracket S \rrbracket_k$ is a k-spanner of $\llbracket G \rrbracket_k$ with at most $n - 1 + O(\frac{m \log n}{k})$ edges, and it takes $\tilde{O}(m)$ work and $\tilde{O}(k)$ time.

Proof. We say that `Ultraspanner` *fails* if on any iteration α, some δ_v in the computation of `ESTCluster` satisfies $\delta_v > k/6$. Observe that if the algorithm does not fail, then every call to `ESTCluster` takes $\tilde{O}(k)$ time (and $\tilde{O}(m)$ work) and returns clusters with diameter at most $2 \cdot k/6 = k/3$. We now bound the probability of failure: for any given vertex v in some Γ_α, the probability that $\delta_v > k/6$ is $e^{-\beta k/6} = e^{-(C/6)\ln n} = n^{-C/6}$. There are at most n vertices in each Γ_α, and at most $\lfloor \log_k W \rfloor + 1 = O(1)$ many iterations since G has bounded aspect ratio, so taking a union bound, the failure probability is at most $O(n^{-C/6+1})$.

Assume now that `Ultraspanner` does not fail. Then, we prove by induction on α that the diameter of each connected component of H_α is at most $k^{\alpha+1}$. This is trivial for $\alpha = 0$, and for $\alpha > 0$, suppose by induction that the statement is true for $\alpha - 1$. Since the algorithm does not fail, `ESTCluster` returns clusters with (unweighted) diameter at most $k/3$. In the weighted Γ_α, these clusters have diameter at most $k/3 \cdot k^\alpha = k^{\alpha+1}/3$. Observe that H_α is formed by starting with these clusters and "uncontracting" each vertex into a component of $H_{\alpha-1}$. By induction, each component in $H_{\alpha-1}$ has diameter at most k^α. Therefore, between any two vertices in a common component of H_α, there is a path between them consisting of at most $k/3$ edges of length k^α and at most $k/3+1$ subpaths each of length at most k^α, each inside a component in $H_{\alpha-1}$. Altogether, the total distance is at most $k/3 \cdot k^\alpha + (k/3+1)\cdot k^\alpha \le k^{\alpha+1}$ (assuming $k \ge 3$).

Let us now argue that the stretch of $\llbracket S \rrbracket_k$ is at most k if the algorithm does not fail. Observe that the edges added to $\llbracket S \rrbracket_k$ in line 9 on an iteration α are precisely the edges whose endpoints belong to different clusters in the corresponding `ESTCluster` call. Conversely, any edge $e \in E_\alpha$ not added to $\llbracket S \rrbracket_k$ have both endpoints in the same cluster. By the previous argument, this cluster has diameter at most $k^{\alpha+1}$ assuming that the algorithm does not fail. Therefore, the stretch of edge e is at most k.

Finally, we bound the number of edges in the output S. By Lemma 7.6.2, for the `ESTCluster` call on iteration α, every edge in E_α has its endpoints in different clusters with probability at most β, which is when it is added to $\llbracket S \rrbracket_k$ in line 9. Over all iterations α, the expected number of edges added to $\llbracket S \rrbracket_k$ in line 9 is at most βm. By Markov's inequality, with probability at least 1/2, there are at most $2\beta m$ edges added in line 9. If this is not the case, we also declare our algorithm to fail. Note that the failure probability now becomes at most $1/2 + O(n^{-C/6+1}) \le 2/3$ (for C large enough). Moreover, it is easy to see that the edges added to S on line 7 form a forest, so at most $n-1$ are added there. Altogether, if the algorithm succeeds, then there are at most $(n-1) + 2\beta m = (n-1) + O(\frac{m \log n}{k})$ edges in the output, which is a k-spanner.

Lastly, `Ultraspanner` can clearly be implemented to run in $\tilde{O}(m)$ work and $\tilde{O}(k)$ time. Moreover, we only need to repeat it $O(\log n)$ times before it succeeds w.h.p. □

Using Lemma 7.6.3, we now prove Lemma 7.3.14 as follows. Let $S \subseteq E$ be the corresponding spanner in G sharing the same edges as $\llbracket S \rrbracket_k \subseteq \llbracket E \rrbracket_k$ (with possibly different weights). Intuitively, since $\llbracket G \rrbracket_k$ approximates the edge weights of G up to factor k, the k-spanner S should be a k^2-spanner on G. More formally, given an edge $(u,v) \in E$, let $u = v_0, v_1, \ldots, v_\ell = v$ be the shortest path in $\llbracket S \rrbracket_k$. We have

$$\begin{aligned} d_S(u,v) \le \sum_{i=1}^{\ell} d_S(v_{i-1}, v_i) \le \sum_{i=1}^{\ell} k \cdot d_{\llbracket S \rrbracket_k}(v_{i-1}, v_i) = k \cdot d_{\llbracket S \rrbracket_k}(u,v) \le k \cdot k d_{\llbracket G \rrbracket_k}(u,v) \\ \le k^2 d_G(u,v). \end{aligned}$$

Therefore, S is a k^2-spanner of G.

7.7 Sherman's Framework via Multiplicative Weights

In this section, we provide a self-contained proof of Theorem 7.4.2 using the *multiplicative weights updates* framework at the loss of an additional $\log(n/\epsilon)$, which can be disregarded in our parallel algorithms.

Theorem 7.7.1: Weaker version of Theorem 7.4.2

Given a transshipment problem, suppose we have already computed a matrix R satisfying:

1. For all demand vectors $b \in \mathbb{R}^n$,

$$\mathsf{opt}(b) \le \mathbb{1}Rb \le \kappa \cdot \mathsf{opt}(b) \tag{7.10}$$

2. Matrix-vector products with R and R^T can be computed in M work and $\text{polylog}(n)$ time[a]

Then, for any transshipment instance with demand vector b, we can compute a flow vector f and a vector of potentials $\tilde{\phi}$ in $\tilde{O}(\kappa^2(m+n+M)\,\epsilon^{-2})$ time that satisfies:

1. $\|Cf\|_1 \le (1+\epsilon)b^T\tilde{\phi} \le (1+\epsilon)\,\mathsf{opt}(b)$
2. $\mathsf{opt}(Af - b) \le \beta\,\mathsf{opt}(b)$

[a] M can potentially be much lower than the number of nonzero entries in the matrix R if it can be efficiently compressed.

We begin with the following classical result on solving linear programs approximately with multiplicative weights update framework. We state it without proof, since the result is a staple in advanced algorithms classes.

Theorem 7.7.2: Solving LPs with Multiplicative Weights Update

Let $\delta \le 1$ and $\omega > 0$ be parameters. Consider a convex set $K \subseteq \mathbb{R}^n$, a matrix $M \in \mathbb{R}^{m\times n}$, and a vector $c \in \mathbb{R}^m$. (We want to investigate approximate feasibility of the set $\{y \in K : My \le c\}$.) Let $\mathcal{O}$ be an oracle that, given any vector $p \in \Delta_m$, either outputs a vector $y \in K$ satisfying $p^T My \le p^T c$ and $\|My - c\|_\infty \le \omega$, or determines that the set $\{y \in K : p^T My \le p^T c\}$ is infeasible. Then, consider the following algorithm:

1. Set $p_0 \leftarrow \frac{1}{m}\mathbb{1} \in \Delta_m$
2. For $t = 1, 2, \ldots, T$ where $T = O(\omega^2 \log m/\delta^2)$:
   (a) Call oracle $\mathcal{O}$ with $p_{t-1} \in \Delta_m$, obtaining vector $y^{(t)}$
   (b) If $\mathcal{O}$ determines that no $y \in K$ exists, then output p_{t-1} and exit
   (c) For each $j \in [m]$, set $w_j^{(t)} \leftarrow w_j^{(t-1)} \cdot \exp(\delta \cdot (My^{(t)} - c)_j) = \exp(\delta \cdot \sum_{i\in[t]}(My^{(i)} - c)_j)$
   (d) For each $j \in [m]$, set $p_j^{(t)} \leftarrow w_j^{(t)} / \sum_{i\in[m]} w_i^{(t)}$
3. Output $\frac{1}{T}\sum_{i\in[T]} y^{(i)} \in K$

If this algorithm outputs a vector $p \in \Delta_m$ on step 2(b), then the set $\{x \in K : Mx \le c\}$ is infeasible. Otherwise, if the algorithm outputs a vector $x \in K$ on step 3, then we have $Mx \le c + \delta\mathbb{1}$.

Lemma 7.7.3

Consider a transshipment instance with demands b and a parameter $t \geq \mathsf{opt}(b)/2$. Let r be a parameter, and let $R \in \mathbb{R}^{[r]\times V}$ be a matrix satisfying (7.10) for some parameter κ. Then, there is an algorithm that performs $O((\kappa/\epsilon)^2 \log n)$ matrix-vector multiplications with A, A^T, R, and R^T, as well as $O((\kappa/\epsilon)^2 \log n)$ operations on vectors in $\mathbb{R}^r$, and outputs either

1. an acyclic flow f satisfying $\mathbb{1}Cf \leq t$ and $\mathbb{1}RAf - Rb < \epsilon t$, or
2. a potential ϕ with value $b^T\phi = t$.

We will run Multiplicative Weights Update (Theorem 7.7.2) to determine feasibility of the region

$$\{y \in \mathbb{R}^r : \left\|y^T RAC^{-1}\right\|_\infty + \frac{1}{t} y^T Rb \leq -\epsilon \text{ and } \|y\|_\infty \leq 1\},$$

which is modeled off the dual LP formulation for transshipment; this connection will become clearer once Claim 7.7.4 is proved.

We set $c := \epsilon\mathbb{1}$ and $K := \{y \in \mathbb{R}^r : \|y\|_\infty \leq 1\}$ in Theorem 7.7.2. As for matrix M, the constraint $\left\|y^T RAC^{-1}\right\|_\infty + \frac{1}{t} y^T Rb \leq -\epsilon$ can be expanded into

$$\pm y^T RAc_e^{-1}\chi_e + \frac{1}{t} y^T Rb \leq -\epsilon \quad \forall e \in E,$$

so the rows of matrix M consist of the $2m$ vectors $(\pm RAc_e^{-1}\chi_e + \frac{1}{t}Rb)^T$ for each $e \in E$.

We now specify the oracle $\mathcal{O}$. On each iteration, the algorithm of Theorem 7.7.2 computes values $p_e^+, p_e^- \geq 0$ for each $e \in E$ satisfying $\sum_e (p_e^+ + p_e^-) = 1$. The oracle needs to compute a vector y satisfying $\|y\|_\infty \leq 1$ and

$$\sum_{e\in E} \left(p_e^+ \big(y^T RAc_e^{-1}\chi_e + \frac{1}{t} y^T Rb\big) + p_e^- \big(-y^T RAc_e^{-1}\chi_e + \frac{1}{t} y^T Rb\big) \right) \leq -\epsilon.$$

This inequality can be rewritten as

$$y^T \left(RA \sum_{e\in E} c_e^{-1}(p_e^+\chi_e - p_e^-\chi_e) + \frac{1}{t} Rb \right) \leq -\epsilon.$$

Observe that a solution y exists iff

$$\mathbb{1}RA \sum_{e\in E} c_e^{-1}(p_e^+\chi_e - p_e^-\chi_e) + \frac{1}{t} Rb \geq \epsilon, \tag{7.11}$$

and if the inequality is true, then the vector $y := -\text{sign}(RA \sum_{e\in E} c_e^{-1}(p_e^+\chi_e - p_e^-\chi_e) + \frac{1}{t}Rb)$ is a solution, so the oracle outputs it.

We set the error parameter δ to be $\epsilon/(2\kappa)$. Then, either the algorithm of Theorem 7.7.2 outputs p that violates (7.11) on some iteration, or after a number of iterations (depending on ρ, which we have yet to bound), the average of all vectors y computed over the iterations satisfies

$$\pm y^T RAc_e^{-1}\chi_e + \frac{1}{t}y^T Rb \le -\epsilon + \frac{\epsilon}{2} = -\frac{\epsilon}{2} \quad \forall e \in E \quad \iff \quad \left\|y^T RAC^{-1}\right\|_\infty + \frac{1}{t}y^T Rb \le -\frac{\epsilon}{2}. \tag{7.12}$$

First, suppose that the second case holds:

Claim 7.7.4

Suppose Theorem 7.7.2 outputs a vector y satisfying (7.12). Then, we can compute a potential ϕ satisfying condition (2).

Proof. Consider the vector $\phi_0 := -(y^T R)^T$. We have

$$\left\|\phi_0^T AC^{-1}\right\|_\infty - \frac{1}{t}\phi_0^T b \le -\frac{\epsilon}{2\kappa} < 0.$$

In particular, $\frac{1}{t}\phi_0^T b > 0$. Let ϕ be the vector ϕ_0 scaled up so that $\frac{1}{t}\phi^T b = 1$. Then,

$$\left\|\phi^T AC^{-1}\right\|_\infty - \frac{1}{t}\phi^T b < 0 \implies \left\|\phi^T AC^{-1}\right\|_\infty < \frac{1}{t}\phi^T b = 1.$$

Since ϕ satisfies the transshipment dual constraints, it is a potential. Moreover, $\phi^T b = t$, so ϕ satisfies condition (2). □

Let us now consider the first case:

Claim 7.7.5

Suppose Theorem 7.7.2 outputs values $p_e^+, p_e^- \ge 0$ satisfying $\sum_e (p_e^+ + p_e^-) = 1$ and

$$\mathbb{1}RA\sum_{e\in E} c_e^{-1}(p_e^+\chi_e - p_e^-\chi_e) + \frac{1}{t}Rb < \epsilon.$$

Then, we can compute an acyclic flow satisfying condition (1).

Proof. Let us construct a flow $f \in \mathbb{R}^E$ defined as $f := -t\sum_e c_e^{-1}(p_e^+\chi_e - p_e^-\chi_e)$. We have

$$\mathbb{1}Cf = \mathbb{1}{-t}\sum_{e\in E}(p_e^+\chi_e - p_e^-\chi_e) = -t\sum_{e\in E}|p_e^+ - p_e^-| \le t\sum_{e\in E}(p_e^+ + p_e^-) = t$$

and

$$\mathbb{1}{-RAf} + Rb < \epsilon t;$$

it remains to show that f is acyclic. Fix an edge $e = (u, v)$, so that $A\chi_e = \chi_u - \chi_v$. We have that f flows from u to v iff

$$f_e > 0 \iff -tc_e^{-1}(p_e^+ - p_e^-) > 0 \iff p_e^+ < p_e^-.$$

By the construction of $p_e^{\pm}$ from Theorem 7.7.2,

$$\begin{aligned} p_e^+ < p_e^- &\iff \sum_{i\in[t]} \big((RAc_e^{-1}\chi_e + \frac{1}{t}Rb)^T y^{(i)} - \epsilon\big) < \sum_{i\in[t]} \big((-RAc_e^{-1}\chi_e + \frac{1}{t}Rb)^T y^{(i)} - \epsilon\big) \\ &\iff (RAc_e^{-1}\chi_e)^T \sum_{i\in[t]} y^{(i)} < (-RAc_e^{-1}\chi_e)^T \sum_{i\in[t]} y^{(i)} \end{aligned}$$

Let $\overline{y} := \sum_{i\in[t]} y^{(i)}$, so that this becomes

$$\begin{aligned} (RAc_e^{-1}\chi_e)^T\overline{y} < (-RAc_e^{-1}\chi_e)^T\overline{y} &\iff \overline{y}^T Rc_e^{-1}(\chi_u - \chi_v) < -\overline{y}^T Rc_e^{-1}(\chi_u - \chi_v) \\ &\iff \overline{y}^T Rc_e^{-1}\chi_u < \overline{y}^T Rc_e^{-1}\chi_v. \end{aligned}$$

Therefore, f will only flow from u to v if $(\overline{y}^T R)_u < (\overline{y}^T R)_v$; it follows that such a flow cannot produce any cycles. □

We now bound the value ω needed for Theorem 7.7.2, which in turn bounds the number of iterations T.

Claim 7.7.6

In Theorem 7.7.2, we can set $\omega := 3\kappa$.

Proof. Consider an iteration where Theorem 7.7.2 outputs values p_e^+, p_e^-, and define $f := -t\sum_e c_e^{-1}(p_e^+\chi_e - p_e^-\chi_e)$ as before, so that $\mathbb{1}Cf \le t$. Let f^* be an optimal flow for demand vector b, and by assumption, $\mathbb{1}Cf^* = \mathsf{opt}(b) \le 2t$. We have

$$\begin{aligned} \mathbb{1}RAf - Rb &= \mathbb{1}RAf^* - Rb + RA(f - f^*) \\ &= \mathbb{1}RA(f - f^*). \end{aligned}$$

The flow $f - f^*$ has cost $\mathbb{1}C(f - f^*) \le \mathbb{1}Cf + \mathbb{1}Cf^* \le t + 2t = 3t$ and satisfies demand $A(f - f^*)$. By (7.10),

$$\mathbb{1}RAf - Rb = \mathbb{1}RA(f - f^*) \le \kappa \cdot \mathsf{opt}(A(f - f^*)) \le \kappa \cdot 3t,$$

which implies that

$$\mathbb{1}RA\sum_{e\in E} c_e^{-1}(p_e^+\chi_e - p_e^-\chi_e) + \frac{1}{t}Rb \le 3\kappa.$$

Since any vector y the algorithm chooses must satisfy $\|y\|_\infty \leq 1$, we must have

$$\left| y^T \left(RA \sum_{e \in E} c_e^{-1}(p_e^+ \chi_e - p_e^- \chi_e) + \frac{1}{t} Rb \right) \right| \leq 3\kappa,$$

so $\omega = 3\kappa$ works, as promised. □

Remark 7.7.7

We only used the upper bound in (7.10); the lower bound will become useful when we work with condition (1) in the lemma. Moreover, we will not need that the flow f is acyclic, but this property may be useful in other applications.

We now claim that Theorem 7.7.1 follows from Lemma 7.7.3. We apply Lemma 7.7.3 $O(\log(n/\epsilon))$ times by binary-searching on the value of t, which we ensure is always at least $\mathsf{opt}(b)/2$ as required by Lemma 7.7.3. Begin with $t = \text{poly}(n)$, an upper bound on $\mathsf{opt}(b)$. First, while Lemma 7.7.3 outputs a flow f (case (1)), set $t \leftarrow \frac{1+\epsilon}{2} t$ for the next iteration. We claim that the new $t' := \frac{1+\epsilon}{2} t$ still satisfies $t' \geq \mathsf{opt}(b)/2$:

$$\mathsf{opt}(b) \leq \mathbb{1}f + \mathsf{opt}(Af - b) \leq \mathbb{1}f + \mathbb{1}RAf - Rb \leq t + \epsilon t \implies \frac{\mathsf{opt}(b)}{2} \leq \left(\frac{1+\epsilon}{2}\right) t = t'.$$

Repeat this until Lemma 7.7.3 outputs a potential ϕ instead, which signifies that $\mathsf{opt}(b) \geq b^T \phi = t$. At this point, we know that $\mathsf{opt}(b) \in [t, 2t]$, and we can properly run binary search in this range, where a flow f returned for parameter t signifies that $\mathsf{opt}(b) \leq t + \epsilon t$, and a potential ϕ returned means that $\mathsf{opt}(b) \geq t$. Through binary search, we can compute two values t_ℓ, t_r such that $t_r - t_\ell \leq \epsilon\, \mathsf{opt}(b)$ and $\mathsf{opt}(b) \in (t_\ell, t_r)$.

Then, run Lemma 7.7.3 with parameter $t = t_\ell/(1+\epsilon)$. We claim that we must obtain a potential ϕ, not a flow f: if we obtain a flow f instead, then

$$\mathsf{opt}(b) \leq \mathbb{1}f + \mathsf{opt}(Af - b) \leq \mathbb{1}f + \mathbb{1}RAf - Rb \leq t + \epsilon t = t_\ell,$$

contradicting the assumption that $\mathsf{opt}(b) > t_\ell$. This potential ϕ satisfies $b^T\phi = \frac{t_\ell}{1+\epsilon} \geq \frac{t_r - \epsilon\, \mathsf{opt}(b)}{1+\epsilon} \geq \frac{\mathsf{opt}(b) - \epsilon\, \mathsf{opt}(b)}{1+\epsilon} = (1 - O(\epsilon))\mathsf{opt}(b)$, which is almost optimal. At this point, we can compute a $(1+\epsilon)$-approximation of $\mathsf{opt}(b)$, but we still need a transshipment flow f.

Next, run Lemma 7.7.3 with parameter $t = t_r$; we claim that we must obtain a flow f this time: if we obtain a potential ϕ instead, then $\mathsf{opt}(b) \geq b^T\phi = t_r$, contradicting the assumption that $\mathsf{opt}(b) < t_r$. This flow satisfies $\mathbb{1}f \leq t_r \leq t_\ell + \epsilon\, \mathsf{opt}(b) \leq \mathsf{opt}(b) + \epsilon\, \mathsf{opt}(b) = (1+\epsilon)\mathsf{opt}(b)$. However, we are not done yet, since f does not satisfy $Af = b$; rather, we only know that $\mathsf{opt}(Af - b) \leq \mathbb{1}R(Af - b) \leq \epsilon\, \mathsf{opt}(b)$. The key idea is to solve transshipment again with demands $b_1 := Af - b$; if we can obtain a $(1+\epsilon)$-approximate flow f_1 satisfying $\mathbb{1}f_1 \leq \mathsf{opt}(b_1) = \mathsf{opt}(Af - b) \leq \epsilon\, \mathsf{opt}(b)$ and $\mathbb{1}R(Af_1 - b_1) \leq \epsilon\, \mathsf{opt}(b_1) \leq \epsilon^2 \mathsf{opt}(b)$, then the composed flow $f + f_1$ has cost

$$\mathbb{1}f + f_1 \leq \mathbb{1}f + \mathbb{1}f_1 \leq (1+\epsilon)\mathsf{opt}(b) + (1+\epsilon)\mathsf{opt}(b_1) \leq (1+\epsilon)\mathsf{opt}(b) + (1+\epsilon)\epsilon\,\mathsf{opt}(b),$$

which is $(1+O(\epsilon))\mathsf{opt}(b)$. We can continue this process, defining $b_i := Af_{i-1} - b_{i-1}$ and computing a flow f_i satisfying $\mathbb{1}f_i \leq \mathsf{opt}(b_i) = \mathsf{opt}(Af_{i-1} - b_{i-1}) \leq \epsilon\,\mathsf{opt}(b_{i-1}) \leq \epsilon^i\mathsf{opt}(b)$ and $\mathbb{1}R(Af_i - b_i) \leq \epsilon\,\mathsf{opt}(b_i) \leq \epsilon^{i+1}\mathsf{opt}(b)$ and adding it on to $f + f_1 + f_2 + \cdots + f_{i-1}$. Assuming $\epsilon \leq 1/2$, say, we can stop after $T := O(\log n)$ iterations, so that the leftover demands b_{T+1} satisfies $\mathsf{opt}(b_{T+1}) \leq \frac{1}{n^3}\mathsf{opt}(b)$. At this point, we can simply run an n-approximate algorithm to demands b_{T+1} by routing through a minimum spanning tree (see Remark 7.4.4), computing a flow f_{T+1} satisfying $\mathbb{1}f_{T+1} \leq n \cdot \frac{1}{n^3}\mathsf{opt}(b) \leq \epsilon\,\mathsf{opt}(b)$, assuming $\epsilon \geq 1/n^2$. (If $\epsilon = O(1/n^2)$, then a transshipment algorithm running in time $\tilde{O}(1/\epsilon^2) \geq \tilde{O}(n^4)$ is trivial.) The final flow $f + f_1 + f_2 + \cdots + f_{T+1}$ has cost at most $(1+O(\epsilon))\mathsf{opt}(b)$.

We have thus computed ϕ satisfying $\mathbb{1}f \leq (1+O(\epsilon))\mathsf{opt}(b) \leq (1+O(\epsilon))b^T\phi$, so (f,ϕ) is an $(1+O(\epsilon))$-approximate flow-potential pair. Finally, we can reset ϵ a constant factor smaller to obtain a $(1+\epsilon)$-approximation. This concludes the algorithm of Theorem 7.7.1; it remains to bound the running time.

In each iteration of Theorem 7.7.2, we perform $O(1)$ matrix-vector multiplications with A, A^T, R, and R^T, as well as an additional $O(m)$ work and polylog(m) time, and the same holds for the oracle $\mathcal{O}$. This requires $O(n + m + M)$ total work. By Theorem 7.7.2, there are $O(\omega^2 \log m/\delta^2) = O((\kappa/\epsilon)^2 \log n)$ iteration inside Lemma 7.7.3 to compute one flow f or potential ϕ. Finally, Lemma 7.7.3 is called polylog(n) times as described above, hence the promised running time.

7.8 Transshipment to Expected SSSP: Sequential

In this section, we devise an algorithm that solves the approximate *expected* single-source shortest path problem, defined below, using multiple *sequential* calls to approximate transshipment. The fact that the recursive calls are made sequentially does not immediately imply a parallel algorithm, but in Section 7.8.1, we show how to save enough computation between the recursive calls to ensure a parallel algorithm. This extra step is more technical than insightful, hence its deferral to a separate subsection.

Finally, in Section 7.9, we show how to reduce SSSP to this expected version of SSSP [14]. Together, Sections 7.8 and 7.9 form a complete proof of Theorem 7.3.7. We remark that while Section 7.9 is simply a rephrasing of a similar routine in [14] and only included for self-containment, this section is novel, albeit still inspired by [14].

Definition 7.8.1: Approximate expected s-SSSP Tree

Given a graph $G = (V, E)$, a source s, and a demand vector b satisfying $b_v \geq 0$ for all $v \neq s$, an *α-approximate expected s-SSSP tree* is a randomized (not necessarily spanning) tree T satisfying

$$\mathbb{E}\left[\sum_{v:b_v>0} b_v \cdot d_T(s,v)\right] \leq \alpha \sum_{v:b_v>0} b_v \cdot d_G(s,v).$$

If $b_v > 0$ for all $v \neq s$, then the tree T must in fact be spanning. We also remark that the term *expected* has two meanings here. First, the tree T is randomized, so the guarantee

$$\sum_{v:b_v>0} b_v \cdot d_T(s,v) \leq \alpha \sum_{v:b_v>0} b_v \cdot d_G(s,v)$$

is only satisfied in expectation. However, even if this guarantee is satisfied with probability 1, the distances $d_T(s, v)$ are not automatically α-approximate distances for *every* v; rather, they only hold on average (weighted by b_v). Note that in the exact setting $\alpha = 1$, all distances $d_T(s, v)$ are indeed exact, but this property breaks down as soon as $\alpha > 1$.

Define $W_{\text{ESSSP}}(n, m, \alpha)$ as the work to compute an α-approximate expected SSSP with arbitrary demand vector b satisfying $b_v \geq 0$ for all $v \neq s$. Our algorithm `ESSSP` is itself recursive and satisfies the following recursion:

Lemma 7.8.2

$W_{\text{ESSSP}}(n, m, (1 + 3\epsilon)\alpha) \leq W_{\text{TS}}(m, \epsilon) + W_{\text{ESSSP}}(n/2, m, \alpha)$.

Of course, we can solve expected SSSP exactly ($\alpha = 1$) in constant time on constant-sized graphs, so this recursion has depth at most $\log_2 n$. Unraveling this recursion, the algorithm calls transshipment at most $\log_2 n$ times, and the error $(1 + 3\epsilon)$ blows up multiplicatively over each recursion level, obtaining

$$W_{\text{ESSSP}}(n, m, (1 + 3\epsilon)^{\log_2 n}) \leq \log_2 n \cdot W_{\text{TS}}(m, \epsilon) + \tilde{O}(m).$$

Resetting the value ϵ, we can rewrite it as

$$W_{\text{ESSSP}}(n, m, 1 + \epsilon) \leq O(\log n) \cdot W_{\text{TS}}(m, \Theta(\epsilon/\log n)) + \tilde{O}(m), \tag{7.13}$$

which is our targeted recursion for our algorithm `ESSSP` below.

Throughout the section, fix a $(1 + \epsilon)$-approximate transshipment flow f satisfying the given demand vector b (with $b_v \geq 0$ for all $v \neq s$). The key insight in our analysis is to focus on a *random walk* based on a slight modification of the transshipment flow f. Define a digraph $\overrightarrow{G} = (V \cup \{\perp\}, \overrightarrow{E}, \vec{w})$ as follows: start from G by bidirecting each edge of E,

Algorithm 12 `ESSSP`$(G = (V, E), s, b, (1 + 3\epsilon)\alpha)$

```
Assumption: demand vector $b$ satisfies $b_s > 0$ and $b_t \le 0$ for all $t \in V \setminus s$
1: Compute a $(1 + \epsilon)$-approximate transshipment $f$ on $G$ with demand vector $b$
2: Initialize the digraph $\overrightarrow{A} \leftarrow \emptyset$
3: Every vertex $u \in V \setminus s$ with $\text{in}(u) \neq \emptyset$ independently samples a random neighbor
   $v \in \text{out}(u)$ with probability $\vec{f}(u,v)/\vec{f}_{\text{out}}(u)$ and adds arc $(u,v)$ to $\overrightarrow{A}$
4: Add a self-loop $(s,s)$ of zero weight to $\overrightarrow{A}$
5: Let $A$ be the undirected version of $\overrightarrow{A}$
6: Initialize $G' \leftarrow (\emptyset, \emptyset)$ as an empty undirected graph    ▷ Graph to be recursed on, with
   $\le n/2$ vertices
7: Initialize $b'$ as an empty vector    ▷ Demands to be recursed on
8: for each connected component $C$ of $A$ do
9:     $c(C) \leftarrow$ total weight of edges in the (unique) cycle in $C$ (possibly the self-loop $(s,s)$)
10:    Let $T_C$ be the graph $C$ with its (unique) cycle contracted into a single vertex $r_C$    ▷
       $T_C$ is a tree
11:    Add a vertex $v_C$ to $G'$, and set demand $b'_{v_C} \leftarrow \sum_{v \in V(C)} b_v$
12: for each edge $(u,u')$ in $E$ do
13:     Let $C$ and $C'$ be the connected components of $A$ containing $u$ and $u'$, respectively
14:     if $C \neq C'$ then
15:         Add an edge between $v_C$ and $v_{C'}$ with weight $w(u,u') + d_{T_C}(u, r_C) + d_{T_{C'}}(u', r_{C'}) +$
            $c(C) + c(C')$
16: Let $s' \leftarrow v_{C_s}$, where $C_s$ is the component of $A$ containing $s$
17: Collapse parallel edges of $G'$ by only keeping the parallel edge with the smallest weight
18: Recursively call ESSSP$(G', s', b', \alpha)$, obtaining an $\alpha$-approximate expected SSSP tree $T'$
    of $G'$
19: Initialize $T \leftarrow \emptyset$    ▷ The expected SSSP tree
20: for each edge $(v,v')$ in $T'$ do
21:     Let $(u,u') \in E$ be the edge responsible for adding edge $(v,v')$ to $G'$
22:     Add edge $(u,u')$ to $T$
23: for each connected component $C$ of $A$ do
24:     Remove an arbitrary edge from the (unique) cycle inside $C$, and add the resulting
        tree to $T$
25: return $T$
```

keeping the same weight in both directions. Then, add a new vertex $\perp$ and a single arc $(s, \perp)$ of weight 0. Let $\overrightarrow{C}$ denote the diagonal matrix indexed by $\overrightarrow{E}$, where diagonal entry $\overrightarrow{C}_{(u,v),(u,v)}$ is the cost of arc $(u, v) \in \overrightarrow{E}$ under the weights $\vec{w}$.

Next, define a flow $\vec{f}$ on digraph $\overrightarrow{G}$ as follows: for each edge $(u, v) \in E$, if $f_{(u,v)} > 0$, then add $f_{(u,v)}$ flow to $\vec{f}$ along the arc (u, v), and if $f_{(u,v)} < 0$, then add $-f_{(u,v)}$ flow to $\vec{f}$ along the arc (v, u). Lastly, add $-b_s$ flow to $\vec{f}$ along arc $(s, \perp)$. Observe that this flow satisfies the demands b_v for each $v \in V \setminus s$, demand 0 for s, and demand b_s for $\perp$. Moreover, the cost $\mathbb{1}\overrightarrow{C}\vec{f}$ of the flow $\vec{f}$ equals $\mathbb{1}Cf$.

For each vertex $v \in V$, define $\text{in}(v) := \{u \in V : \vec{f}(u, v) > 0\}$ as the neighbors of v that send flow to v, and define $\text{out}(v) := \{u \in V : \vec{f}(v, u) > 0\}$ as the neighbors of v that receive flow from v. For convenience, define $\vec{f}_{\text{in}}(v) := \sum_{u \in \text{in}(v)} \vec{f}(u, v)$ and $\vec{f}_{\text{out}}(v) := \sum_{u \in \text{out}(v)} \vec{f}(v, u)$.

Define $b^+(v) := \max\{b_v, 0\}$, so that $b^+(s) = 0$ and $b^+(v) = b(v)$ for all $v \neq s$. Define $V^* \subseteq V$ as the vertices t for which there exists a $t \to \perp$ path using only arcs supported by the flow $\vec{f}$ (that is, arcs (u, v) with $\vec{f}(u, v) > 0$). We will only be considering vertices in V^* for the rest of this section.

Claim 7.8.3

For all vertices $t \in V$, if $b(t) = b^+(t) > 0$, then $t \in V^*$. Moreover, for all $v \in V^* \cup \{\perp\}$, we have $\text{in}(v) \subseteq V^* \cup \{\perp\}$ and $\text{out}(v) \in V^* \cup \{\perp\}$. In other words, the vertices in $V^* \cup \{\perp\}$ are separated from the vertices in $V \setminus V^*$ by arcs supported by $\vec{f}$.

Proof. For the first claim, let $R \subseteq V \cup \{\perp\}$ be all vertices reachable from t along arcs supported by $\vec{f}$; we need to show that $\perp \in R$. Since there are no arcs in $\vec{f}$ going out of R, by conservation of flow, we must have $\sum_{v \in R} b(v) \leq 0$. Since $t \in R$ and $b^+(t) = b(t) > 0$, we have $\sum_{v \in R \setminus t} b(v) < 0$. But the only vertex in $V \cup \{\perp\}$ with negative demand is $\perp$, so it must hold that $\perp \in R$, as desired.

We now prove the second statement. If suffices to show that there are no arcs between $V^* \cup \{\perp\}$ and $V \setminus V^*$. There cannot be an arc (u, v) supported by $\vec{f}$ with $u \notin V^* \cup \{\perp\}$ and $v \in V^* \cup \{\perp\}$, since that would mean u can reach $\perp$ by first traveling to v. Suppose for contradiction that there is an arc from $V^* \cup \{\perp\}$ to $V \setminus V^*$. Since there is no arc the other way, by conservation of flow, it must follow that $\sum_{v \notin V^* \cup \{\perp\}} b(v) < 0$. But the only vertex $\perp$ with negative demand is $\perp$, so it cannot be that $\sum_{v \notin V^* \cup \{\perp\}} b(v) < 0$, a contradiction. $\square$

For each vertex $t \in V^*$, consider the natural random walk from t to $\perp$ in $\overrightarrow{G}$, weighted by the flow $\vec{f}$: start from t, and if the walk is currently at vertex $u \in V$, then travel to vertex $v \in \text{out}(v)$ with probability $\vec{f}(u, v)/\vec{f}_{\text{out}}(u)$ (independent of all previous steps); stop when $\perp$ is reached. Let this random walk be the random variable W_t, which is guaranteed to stay within $V^* \cup \{\perp\}$ by Claim 7.8.3.

Given a walk W in $\overrightarrow{G}$, define $\mathsf{length}(W)$ to be the length of the walk under the weights $\vec{w}$. In particular, $\mathsf{length}(W_t)$ is the length of the random walk W_t from t to $\perp$.

The claim below relates the transshipment cost to the expected lengths of the random walks W_t for $t \in V^*$. This allows us to later charge our expected SSSP distances to the lengths of these concrete random walks, which are easier to work with than the transshipment flow itself.

Claim 7.8.4

We have
$$\sum_{t \in V^*} b^+(t) \cdot \mathbb{E}[\mathsf{length}(W_t)] \le \mathbb{1}\overrightarrow{C}\vec{f}.$$
In other words, if, for each $t \in V^*$, we sample a random walk from t to $\perp$ and multiply its length by $b^+(t)$, then the sum of the multiplied lengths over all t is at most $\mathbb{1}\overrightarrow{C}\vec{f}$ in expectation.

Proof. For each vertex $v \in V^* \cup \{\perp\}$, let $\mathsf{freq}_t(v)$ be the expected number of times v appears in W_t. We first prove the following:

Subclaim 7.8.5

For all vertices $v \in V^*$, $\sum_{t \in V^*} b^+(t) \cdot \mathbb{E}[\mathsf{freq}_t(v)] = \vec{f}_{\text{out}}(v)$.

Proof. For each $t \in V^*$, treat $\mathbb{E}[\mathsf{freq}_t(v)]$ as a function from $V^* \cup \{\perp\}$ to $\mathbb{R}_{\ge 0}$, which satisfies the following equations:

$$\mathbb{E}[\mathsf{freq}_t(t)] = 1 + \sum_{u \in \text{in}(t)} \mathbb{E}[\mathsf{freq}_t(u)] \cdot \frac{\vec{f}(u,t)}{\vec{f}_{\text{out}}(u)} \tag{7.14}$$

$$\mathbb{E}[\mathsf{freq}_t(v)] = \sum_{u \in \text{in}(v)} \mathbb{E}[\mathsf{freq}_t(u)] \cdot \frac{\vec{f}(u,v)}{\vec{f}_{\text{out}}(u)} \qquad \forall v \in (V^* \cup \{\perp\}) \setminus t \tag{7.15}$$

$$\mathbb{E}[\mathsf{freq}_t(\perp)] = 1$$

Note that the equations are well-defined, since by Claim 7.8.3, all vertices $u \in \text{in}(t)$ are in V^* if $t \in V^*$. Define the function $f(v) := \sum_{t \in V^*} b^+(v) \cdot \mathbb{E}[\mathsf{freq}_t(v)]$, so our goal is to show that $f(v) = \vec{f}_{\text{out}}(v)$ for each $v \in V^*$. We sum Equations (7.14) and (7.15) as follows: for each t, multiply Equations (7.14) and (7.15) by $b^+(t)$, and then sum over all t, obtaining

$$f(v) = b^+(v) + \sum_{u \in \text{in}(t)} f(u) \cdot \frac{\vec{f}(u,t)}{\vec{f}_{\text{out}}(u)} \qquad \forall v \in V^* \cup \{\perp\} \tag{7.16}$$

$$f(\perp) = \sum_{t \in V^*} b^+(t) \tag{7.17}$$

We first show that the solution $f(v) = \vec{f}_{\text{out}}(v)$ for $v \in V^*$ and $f(\perp) = \sum_{t \in V^*} b^+(t)$ satisfies the above system of equations. Equation (7.17) is clearly satisfied, and for (7.16), we have

$$\vec{f}_{\text{out}}(v) = b^+(v) + \vec{f}_{\text{in}}(v) = b^+(v) + \sum_{u \in \text{in}(v)} \vec{f}(u,v) = b^+(v) + \sum_{u \in \text{in}(v)} \vec{f}_{\text{out}}(u) \cdot \frac{\vec{f}(u,v)}{\vec{f}_{\text{out}}(u)}.$$

We now claim that there is a unique solution to f, which is enough to prove the claim. Suppose there are two solutions f and f' that satisfy (7.16) and (7.17). Let g be their difference: $g(v) := f(v) - f'(v)$ for all $v \in V^*$, which satisfies

$$g(v) = \sum_{u \in \text{in}(t)} g(u) \cdot \frac{\vec{f}(u,t)}{\vec{f}_{\text{out}}(u)} \qquad \forall v \in V^* \cup \{\perp\} \tag{7.18}$$

$$g(\perp) = 0 \tag{7.19}$$

We want to show that $g(v) = 0$ for all $v \in V^*$. It suffices to show that $g(v) \le 0$ for all $v \in V^*$, since Equations (7.18) and (7.19) are satisfied with $g(v)$ replaced by $-g(v)$, so we would prove both $g(v) \le 0$ and $-g(v) \le 0$, which would give $g(v) = 0$.

To show that $g(v) \le 0$ for all $v \in V^*$, let $v^* := \arg\max_{v \in V^*} g(v)$. By (7.18), $g(v^*)$ is a weighted average of $g(u)$ over vertices $u \in \text{in}(v^*)$, so

$$g(v^*) = \sum_{u \in \text{in}(v^*)} g(u) \cdot \frac{\vec{f}(u,t)}{\vec{f}_{\text{out}}(u)} \le \sum_{u \in \text{in}(v^*)} g(v^*) \cdot \frac{\vec{f}(u,t)}{\vec{f}_{\text{out}}(u)} = g(v^*),$$

so the inequality must be satisfied with equality, and $g(u) = g(v^*)$ for all $u \in \text{in}(v^*)$. Continuing this argument, any vertex u for which there exists a (possibly empty) $u \to v^*$ path in $\vec{f}$ satisfies $g(u) = g(v^*)$. Define $R := \{u \in V^* : \text{exists } u \to v^* \text{ path in } \vec{f}\}$ as these vertices. Suppose for contradiction that $g(v^*) > 0$, or equivalently, $g(u) > 0$ for all $u \in R$. Then, summing (7.18) over all $v \in R$, we obtain

$$\begin{aligned}
\sum_{v \in R} g(v) &= \sum_{v \in R} \sum_{u \in \text{in}(v)} g(v) \cdot \frac{\vec{f}(u,v)}{\text{out}(u)} \\
&= \sum_{v \in R} \sum_{u \in \text{in}(v) \cap R} g(v) \cdot \frac{\vec{f}(u,v)}{\text{out}(u)} \\
&= \sum_{u \in R} \left(\frac{g(u)}{\text{out}(u)} \sum_{v \in R : u \in \text{in}(v)} \vec{f}(u,v) \right) \\
&\overset{g(u) \ge 0}{\le} \sum_{u \in R} \left(\frac{g(u)}{\text{out}(u)} \sum_{v \in V^*} \vec{f}(u,v) \right)
\end{aligned} \tag{7.20}$$

$$
\begin{aligned}
&= \sum_{u \in R} \left(\frac{g(u)}{\mathrm{out}(u)} \, \mathrm{out}(v) \right) \\
&= \sum_{u \in R} g(u).
\end{aligned}
$$

Since $v^* \in V^*$, there exists a $v^* \to \perp$ path in $\vec{f}$. Since $v^* \in R$ and $\perp \notin R$, there exists an arc (u', v') on the path with $u' \in R$ and $v' \notin R$ (and $\vec{f}(u', v') > 0$). Consider the inequality at (7.20). The inequality holds for each $u \in R$ in the outer summation, but for $u = u'$ in particular, we have $\sum_{v \in R : u \in \mathrm{in}(v)} \vec{f}(u', v) < \sum_{v \in V^*} \vec{f}(u, v)$. Since $g(u) > 0$ by assumption, the inequality at (7.20) is actually strict, which gives the contradiction $\sum_{v \in R} g(v) < \sum_{u \in R} g(u)$. It follows that $g(v) \le g(v^*) \le 0$ for all $v \in V^*$. ◇

We now resume the proof of Claim 7.8.4. For all $t \in V^*$, by linearity of expectation,

$$
\mathbb{E}[\mathsf{length}(W_t)] = \sum_{u \in V^*} \mathsf{freq}_t(u) \cdot \sum_{v \in \mathrm{out}(u)} \frac{\vec{f}(u,v)}{\vec{f}_{\mathrm{out}}(u)} \, \vec{w}(u,v).
$$

For each $t \in V^*$, multiply the equation by $b^+(t)$, and sum the equations, obtaining

$$
\begin{aligned}
\sum_{t \in V^*} b^+(t) \cdot \mathbb{E}[\mathsf{length}(W_t)] &= \sum_{t \in V^*} b^+(t) \left(\sum_{u \in V^*} \mathsf{freq}_t(u) \cdot \sum_{v \in \mathrm{out}(u)} \frac{\vec{f}(u,v)}{\vec{f}_{\mathrm{out}}(u)} \, \vec{w}(u,v) \right) \\
&= \sum_{u \in V^*} \left(\left(\sum_{t \in V^*} b^+(t) \cdot \mathsf{freq}_t(u) \right) \sum_{v \in \mathrm{out}(u)} \frac{\vec{f}(u,v)}{\vec{f}_{\mathrm{out}}(u)} \, \vec{w}(u,v) \right) \\
&\overset{\text{Sub. 7.8.5}}{=} \sum_{u \in V^*} \left(\vec{f}_{\mathrm{out}}(u) \cdot \sum_{v \in \mathrm{out}(u)} \frac{\vec{f}(u,v)}{\vec{f}_{\mathrm{out}}(u)} \, \vec{w}(u,v) \right) \\
&= \sum_{u \in V^*} \sum_{v \in \mathrm{out}(u)} \vec{f}(u,v) \, \vec{w}(u,v) \\
&\le \sum_{u \in V} \sum_{v \in \mathrm{out}(u)} \vec{f}(u,v) \, \vec{w}(u,v) \\
&= \mathbb{1} \overrightarrow{C} \vec{f}.
\end{aligned}
$$

This concludes Claim 7.8.4. □

We remark that all our claims so far are in expectation, and therefore do not care about dependencies between the walks W_t for different $t \in V^*$.

For each walk W_t, we can imagine sampling it as follows: for each vertex $u \in V^*$, sample an infinite sequence of arcs (u, v) for $v \in \mathrm{out}(u)$, each independent of the others and with probability $\vec{f}(u,v)/\vec{f}_{\mathrm{out}}(u)$; let these arcs be $e_1^u, e_2^u, \ldots$. Once the arcs are sampled for each

$u \in V^*$, the random walk W_t is determined as follows: start at t, and if the walk is currently at $u \in V^*$, then travel along the next *unused* arc in the sequence $e_1^u, e_2^u, \ldots$; in other words, if we have visited vertex u on the walk k times before the current visit, then travel along the arc e_{k+1}^u. It is easy to see that the distribution of this random walk is exactly W_t.

Let us first sample the set of sequences $e_1^u, e_2^u, \ldots$ for each $u \in V^*$, and then determine the walks W_t using this set of sequences for all t; note that the walks W_t are heavily dependent on each other this way. Furthermore, observe that the arcs $\{e_1^u : u \in V^*\}$ are distributed the same way as the arcs in $\overrightarrow{A}$ from ESSSP.

Lemma 7.8.6

Suppose we execute line 3 of ESSSP, so that each vertex u has (independently) sampled a neighbor v^u and added arc (u, v^u) is added to $\overrightarrow{A}$. Let E be the event that in the infinite sequences $e_1^u, e_2^u, \ldots$, we have $e_1^u = (u, v^u)$ for all $u \in V^*$. Let $\mathsf{opt} = \mathsf{opt}_G(b)$ be the optimum transshipment cost with demands b in G, and let $\mathsf{opt}' = \mathsf{opt}_{G'}(b')$ be the optimum transshipment cost in the recursive call at line 18. Then, by conditioning on E in the random walks W_t, we obtain

$$\sum_{t \in V^*} b^+(t) \cdot \mathbb{E}[\mathsf{length}(W_t) \mid E] + 2\epsilon \cdot \mathsf{opt} \geq \mathbb{E}\left[\sum_{t \in V^*} (b^+(t) \cdot d_{T_C}(u, r_C)) \middle| E\right] + \mathsf{opt}'. \tag{7.21}$$

Proof. Consider the following "greedy" cycle-finding algorithm for walks: given a walk W, travel along the walk in the forward direction, and whenever a cycle is found, immediately remove the cycle; output the set of all cycles removed. Let $\mathsf{cycles}(W_t)$ be the total length of all cycles removed, where length is measured by the weights $\vec{w}$. We first show that $\mathsf{cycles}(W_t)$ must be small compared to opt, so that we can later charge to arcs in these cycles.

Subclaim 7.8.7

Given a walk W, let $\mathsf{cycles}(W)$ be the total length of cycles computed by the cycle-finding algorithm, where length is measured by the weights $\vec{w}$. Recall that the transshipment flow $\vec{f}$ is a $(1+\epsilon)$-approximation of the optimum opt (that is, $\mathbb{1}\overrightarrow{C}\vec{f} \leq (1+\epsilon)\mathsf{opt}$); then, we have

$$\sum_{t \in V^*} b^+(t) \cdot \mathbb{E}[\mathsf{cycles}(W_t)] \leq \epsilon \cdot \mathsf{opt}.$$

Proof. For each walk W_t sampled, remove the cycles computed by the algorithm to obtain another walk W_t', still from t to $\perp$. Then, let W_t^- be the walk W_t' minus the last vertex $\perp$, whose new last vertex must be s. The flow obtained by sending $b^+(t)$ flow along the walk W_t^- for each $t \in V^* \setminus s$ is a transshipment flow satisfying the demands $b^+(t)$ for each $t \neq s$.

Therefore,

$$\begin{aligned}
\mathsf{opt} &\le \sum_{t\in V^*\setminus s} b^+(t)\cdot \mathbb{E}[\mathsf{length}(W_t^-)] \\
&= \sum_{t\in V^*\setminus s} b^+(t)\cdot \mathbb{E}[\mathsf{length}(W_t')] \\
&\le \sum_{t\in V^*} b^+(t)\cdot \mathbb{E}[\mathsf{length}(W_t')]
\end{aligned}$$

$$\begin{aligned}
&= \sum_{t\in V^*} b^+(t)\cdot \mathbb{E}[\mathsf{length}(W_t) - \mathsf{cycles}(W_t)] \\
&\le \sum_{t\in V^*} b^+(t)\cdot \mathbb{E}[\mathsf{length}(W_t)] - \sum_{t\in V^*} b^+(t)\cdot \mathbb{E}[\mathsf{cycles}(W_t)] \\
&\overset{\text{Clm. 7.8.4}}{\le} \mathbb{1}\overrightarrow{C}\vec{f} - \sum_{t\in V^*} b^+(t)\cdot \mathbb{E}[\mathsf{cycles}(W_t)] \\
&\le (1+\epsilon)\mathsf{opt} - \sum_{t\in V^*} b^+(t)\cdot \mathbb{E}[\mathsf{cycles}(W_t)],
\end{aligned}$$

and rearranging proves the claim. ◇

Let us return to proving (7.21). We actually show that for *any* set of infinite sequences $\{e_1^u, e_2^u, \dots : u \in V^*\}$ satisfying event E (that is, $e_1^u = (u, v^u)$ for all $u \in V^*$), we have

$$\sum_{t\in V^*} b^+(t)\cdot (\mathsf{length}(W_t) + 2\mathsf{cycles}(W_t)) \ge \sum_{t\in V^*} b^+(t)\cdot d_{T_C}(u, r_C) + \mathsf{opt}'. \tag{7.22}$$

Assuming (7.22), taking expectations and then applying Subclaim 7.8.7 gives

$$\begin{aligned}
&\sum_{t\in V^*} b^+(t)\cdot \mathbb{E}[\mathsf{length}(W_t)\mid E] + 2\epsilon\cdot\mathsf{opt} \\
&\qquad\ge \sum_{t\in V^*} b^+(t)\cdot \mathbb{E}[\mathsf{length}(W_t)\mid E] + 2\sum_{t\in V^*} b^+(t)\cdot \mathbb{E}[\mathsf{cycles}(W_t)\mid E] \\
&\qquad= \sum_{t\in V^*} b^+(t)\cdot (\mathbb{E}[\mathsf{length}(W_t)\mid E] + 2\mathbb{E}[\mathsf{cycles}(W_t)\mid E]) \\
&\qquad\overset{(7.22)}{\ge} \mathbb{E}[b^+(t)\cdot d_{T_C}(u, r_C)\mid E] + \mathsf{opt}',
\end{aligned}$$

as desired.

For the remainder of the proof, we will show (7.22) given *any* set of arbitrary infinite sequences $\{e_1^u, e_2^u, \dots : u \in V^*\}$ satisfying $e_1^u = (u, v^u)$ for all $u \in V^*$. In particular, now that we have fixed these infinite sequences, all randomness goes away, so there are no more probabilistic arguments for the remainder of the proof.

For each t, we can view the random walk W_t as follows: for each vertex $u \in V^*$, the first time the walk reaches u, it must travel along arc (u, v^u) in $\overrightarrow{A}$, and any time after that, it can choose an arbitrary arc in $\mathrm{out}(u)$.

Subclaim 7.8.8

Let C be a connected component of A in `ESSSP` containing a vertex u. If the walk W contains u, then it contains every arc on the (unique, possibly empty) path from u to the (unique) cycle in C (defined as the path from u to the closest vertex on that cycle).

Proof. Consider the first time the walk visits vertex u. Then, the walk must traverse the arc $(u, v^u) = e_1^u$, which is the first arc along the path from u to the cycle in C. We can then repeat the argument with u replaced by v^u, considering the first time the walk visits vertex v^u. Continuing this argument until we arrive at a vertex on the cycle proves the claim. ◇

Subclaim 7.8.9

Let C be a connected component of A in `ESSSP` containing a vertex u. Suppose a walk W contains an arc (u, v) with $v \neq v^u$. Then, the cycles output by the greedy cycle-finding algorithm contains (1) every arc along the (unique, possibly empty) path from u to the (unique) cycle in C, and (2) every arc in the (unique) cycle in C in the direction given by $\overrightarrow{A}$.

Proof. For this proof only, imagine that the walk visits one vertex per unit of time, so we say that the walk reaches a vertex v *at time* i if v is the i'th vertex of the walk.

We first prove statement (2). Consider the first time i that the walk reaches any vertex in C, which must occur since vertex u in C is reached eventually. Then, right after time i, the walk will travel along $\overrightarrow{A}$ towards the unique cycle in C, and then travel along C in the direction given by $\overrightarrow{A}$. The greedy cycle-finding algorithm then removes that cycle, proving statement (2).

We now prove statement (1). If u is inside the cycle of C, then the path is empty and there is nothing to prove. Otherwise, let (u', v') be an arbitrary arc along this (nonempty) path. We first show that (u', v') is traveled at least once in the walk. For this proof only, for two vertices u, v in C (possibly $u = v$), let $P(u, v)$ be the (possibly empty) path in $\overrightarrow{A}$ from u to v (inclusive), which will always exist and be unique when we use it. Consider the first time i that the walk reaches any vertex v in $P(u, u')$ (possibly u or u'). Since the walk visits vertex u at least once, this 'first time i' must occur. Then, right after time i, the walk must travel along $P(v, u')$, and then along arc (u', v'). Let the walk reach vertex u' at time i' (so it reaches v' at time $i' + 1$).

We now show that this first traversal of arc (u', v') at time i' is indeed added to a cycle by the greedy algorithm. Define v as before, and consider the next time j the walk reaches

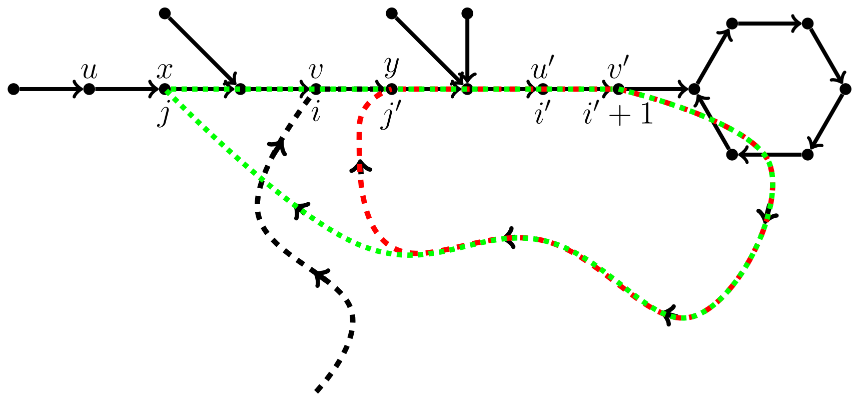

Figure 7.4: The two cases of statement (1) from Subclaim 7.8.9. The curved paths that do not align with the horizontal edges are arbitrary paths in other parts of the graph. The red dashed line marks the cycle for the first case, and the green dotted line marks the cycle for the second case.

any vertex x in $P(u,v)$. Then, right after time j, the walk must travel along $P(x,v)$. Since the walk contains an arc (u,v) with $v \neq v^u$, vertex u is reached at least twice, so this 'next time j' must occur (we need at least *twice* in case $v = u$). By time j, one of two things can happen (see Figure 7.4):

1. The walk returns to some vertex y in $P(v,u')$ before time j (but after time $i'+1$). Let time $j' \in (i'+1, j)$ be the first such return, and let y be the vertex returned to. Before time j', all vertices in $P(v,u')$ were visited exactly once, so either all arcs in $P(v,v')$ were added to the same cycle before time j, or none of them were added to any cycle before time j. In the former case, the path $P(v,v')$ includes arc (u',v'), so we are done. In the latter case, we obtain the cycle consisting of the path $P(y,v')$, followed by the (remaining) arcs in the walk from time $i'+1$ to time j'. The greedy algorithm then adds this cycle, which contains arc (u',v').

2. The walk does not return to any vertex in $P(v,u')$ after time $i'+1$ and before time j. By the same argument as the case above, either all arcs in $P(v,v')$ were added to the same cycle before time j, or none were added to any cycle before time j. Again, in the former case, we are done, so suppose the latter. Right after time j, the walk travels along $P(x,v)$, and no vertex on $P(x,v)$ except possibly v was visited before time j. Therefore, the next time the walk encloses a cycle (after time j) is when it travels along $P(x,v)$ and reaches v. At this point, all arcs in $P(x,v')$ (including arc (u',v')) are added into the cycle that begins and ends at v, so we are done.

Therefore, in both cases, we are done. $\diamond$

Subclaim 7.8.10

For each vertex $t \in V^*$, let C be the component in A containing t. We have

$$\mathsf{length}(W_t) + 2\mathsf{cycles}(W_t) \geq d_{T_C}(t, r_C) + d_{G'}(s, r_C).$$

Proof. Define a *subwalk* of W to be a contiguous subsequence of the walk when it is viewed as a sequence of vertices. We partition the walk W into subwalks as follows. Start with the first vertex t of the walk, and let C be the component of A containing t. Consider the last vertex v of the walk also inside C. If $v = s$, then we are done; otherwise, break of the subwalk from the beginning of the walk to the (last occurrence of) vertex v in the walk, and recursively apply the procedure on the remaining vertices of the walk.

Let the subwalks be $W_1, W_2, \ldots, W_k$ in that order, and for $i \in [k]$, let u_i and v_i be the first and last vertex of W_i, and let C_i be the component of A containing both u_i and v_i. By construction, all components C_i are distinct. For each $i \in [k-1]$, the arc (v_i, u_{i+1}) in $\vec{f}$ is responsible for adding an arc $(v_{C_i}, v_{C_{i+1}})$ in line 15 of weight $w(v_i, u_{i+1}) + d_{T_{C_i}}(v_i, r_{C_i}) + d_{T_{C_{i+1}}}(u_{i+1}, r_{C_{i+1}}) + c(C_i) + c(C_{i+1})$; let this arc be (v'_i, u'_{i+1}). Consider the path from r_C to s' in G' consisting of the edges (v'_i, u'_{i+1}) for $i \in [k-1]$. It suffices to show that this path has length at most $\mathsf{length}(W_t) + 2\mathsf{cycles}(W_t) - d_{T_C}(t, r_C)$; the distance $d_{G'}(s, r_C)$ can only be smaller. In other words, we want to show that

$$\sum_{i=1}^{k-1} \Big(w(v_i, u_{i+1}) + d_{T_{C_i}}(v_i, r_{C_i}) + d_{T_{C_{i+1}}}(u_{i+1}, r_{C_{i+1}}) + c(C_i) + c(C_{i+1}) \Big) \\ \leq \mathsf{length}(W_t) + 2\mathsf{cycles}(W_t) - d_{T_C}(t, r_C). \qquad (7.23)$$

For each $i \in [k-1]$, the distance $d_{T_{C_i}}(v_i, r_{C_i})$ equals the length of the (possibly empty) path P_i in C_i from v_i to the (closest vertex on the unique) cycle in C_i. Since u_{i+1} is not in C_i, we have $u_{i+1} \neq v_1^{u_{i+1}}$, so by Subclaim 7.8.9, every arc in P_i is added to some cycle by the greedy cycle-finding algorithm. Moreover, by Subclaim 7.8.9, every edge in the (unique) cycle in C_i is also added to some cycle. All such arcs mentioned are distinct, so we obtain

$$\sum_{i=1}^{k-1} \Big(d_{T_{C_i}}(v_i, r_{C_i}) + c(C_i) \Big) \leq \mathsf{cycles}(W_t).$$

For the distances $d_{T_{C_{i+1}}}(u_{i+1}, r_{C_{i+1}})$ for $i \in [k-1]$, as well as the distance $d_{T_{C_1}}(u_1, r_{C_1}) = d_{T_C}(t, r_C)$, we charge them to the walk W_t directly. For each $i \in [k]$, since the walk contains vertex u_i in C_i, by Subclaim 7.8.8, it contains all arcs on the path from u_i to the cycle

in C_i, whose weights sum to exactly $d_{T_{C_i}}(u_i, r_{C_i})$. Finally, we also charge the edge weights $w(v_i, u_{i+1})$ to the walk W_t, since the walk contains them by construction, and they are edge-disjoint from each other and all arcs in any C_i. Thus,

$$\begin{aligned}
&\sum_{i=1}^{k-1} \Big(w(v_i, u_{i+1}) + d_{T_{C_i}}(v_i, r_{C_i}) + d_{T_{C_{i+1}}}(u_{i+1}, r_{C_{i+1}}) + c(C_i) + c(C_{i+1}) \Big) \\
&\qquad \le 2 \sum_{i=1}^{k-1} \Big(d_{T_{C_i}}(v_i, r_{C_i}) + c(C_i) \Big) + \sum_{i=1}^{k-1} \Big(w(v_i, u_{i+1}) + d_{T_{C_{i+1}}}(u_{i+1}, r_{C_{i+1}}) \Big) \\
&\qquad \le 2 \sum_{i=1}^{k-1} \Big(d_{T_{C_i}}(v_i, r_{C_i}) + c(C_i) \Big) + \sum_{i=1}^{k-1} w(v_i, u_{i+1}) + \sum_{i=1}^{k} d_{T_{C_i}}(u_i, r_{C_i}) - d_{T_C}(t, r_C) \\
&\qquad \le 2\mathsf{cycles}(W_t) + \mathsf{length}(W_t) - d_{T_C}(t, r_C),
\end{aligned}$$

proving (7.23). $\diamond$

We now resume the proof of Lemma 7.8.6. Multiplying the inequality by $b^+(t)$ for each t and summing over all t gives

$$\sum_{t \in V^*} b^+(t) \cdot \mathsf{length}(W_t) + 2\mathsf{cycles}(W_t) \ge \sum_{t \in V^*} b^+(t) \cdot (d_{T_C}(t, r_C) + d_{G'}(s, r_C)). \tag{7.24}$$

Consider routing, for each component C of A and vertex t in C, $b^+(t)$ amount of flow along the shortest path from s' to v_C in G'. By construction of the demand vector b', this is a transshipment flow that satisfies demands b'. Therefore, the optimum transshipment cost opt' can only be smaller (in fact, it is equal), and we obtain

$$\mathsf{opt}' \le \sum_{t \in V^*} b^+(t) \cdot d_{G'}(s, r_C).$$

Together with (7.24), this concludes (7.22), and hence Lemma 7.8.6. $\square$

Corollary 7.8.11

Over the randomness of `ESSSP` (in particular, the random choices on line 3),

$$\mathbb{E}\left[\sum_{t \in V^*} b^+(t) \cdot d_{T_C}(u, r_C) + \mathsf{opt}' \right] \le \mathbb{1}\overrightarrow{C}\vec{f} + 2\epsilon \cdot \mathsf{opt}.$$

Proof. We apply Lemma 7.8.6 by taking the expectation of (7.21) over the randomness on line 3, which effectively removes the conditioning by E, and obtain

$$\sum_{t \in V^*} b^+(t) \cdot \mathbb{E}[\mathsf{length}(W_t)] + 2\epsilon \cdot \mathsf{opt} \ge \mathbb{E}[b^+(t) \cdot d_{T_C}(u, r_C) + \mathsf{opt}'].$$

This, along with $\sum_{t \in V^*} b^+(t) \cdot \mathbb{E}[\mathsf{length}(W_t)] \le \mathbb{1}\overrightarrow{C}\vec{f}$ from Claim 7.8.4, finishes the proof. $\square$

Claim 7.8.12

Define T' and T as in ESSSP. For each component C of A and each vertex v in C, we have $d_T(s,v) \le d_{T'}(s', v_C) + d_{T_C}(t, r_C)$.

Proof. Follows easily by construction (lines 20 to 24), so proof is omitted. □

Claim 7.8.13

Over the entire randomness of ESSSP (including the recursion at line 18), we have

$$\mathbb{E}\left[\sum_{t\in V} b^+(t)\cdot d_T(s,t)\right] \le \alpha(1+3\epsilon)\mathsf{opt}.$$

Proof. By Claim 7.8.3, we have $b^+(t)=0$ for all $t \in V \setminus V^*$, so it suffices to prove

$$\mathbb{E}\left[\sum_{t\in V^*} b^+(t)\cdot d_T(s,t)\right] \le \alpha(1+3\epsilon)\mathsf{opt}.$$

By recursion, we have the guarantee

$$\mathbb{E}\left[\sum_{C: v_C\neq s'} b'_{v_C}\cdot d_{T'}(s', v_C)\right] \le \alpha\mathsf{opt}.$$

For each vertex t, let C_t be the component of A containing t. We have

$$\begin{aligned}
\mathbb{E}\left[\sum_{t\in V^*} b^+(t)\cdot d_T(s,v)\right] &\overset{\text{Clm. 7.8.12}}{\le} \mathbb{E}\left[\sum_{t\in V^*}\left(b^+(t)\cdot d_{T'}(s', v_{C_t}) + d_{T_{C_t}}(t, r_{C_t})\right)\right]\\
&= \mathbb{E}\left[\sum_{t\in V^*} b^+(t)\cdot d_{T'}(s', v_{C_t})\right] + \mathbb{E}\left[\sum_{t\in V^*} b^+(t)\cdot d_{T_{C_t}}(t, r_{C_t})\right]\\
&= \mathbb{E}\left[\sum_{C: v_C\neq s'} b'_{v_C}\cdot d_{T'}(s', v_C)\right] + \mathbb{E}\left[\sum_{t\in V^*} b^+(t)\cdot d_{T_{C_t}}(t, r_{C_t})\right]\\
&\le \alpha\mathbb{E}[\mathsf{opt}'] + \mathbb{E}\left[\sum_{t\in V^*} b^+(t)\cdot d_{T_{C_t}}(t, r_{C_t})\right]\\
&\le \alpha\left(\mathbb{E}[\mathsf{opt}'] + \mathbb{E}\left[\sum_{t\in V^*} b^+(t)\cdot d_{T_{C_t}}(t, r_{C_t})\right]\right)\\
&\overset{\text{Cor. 7.8.11}}{\le} \alpha\left(\mathbb{1}\vec{C}\vec{f} + 2\epsilon\cdot\mathsf{opt}\right)\\
&\le \alpha((1+\epsilon)\mathsf{opt} + 2\epsilon\cdot\mathsf{opt}) = \alpha(1+3\epsilon)\mathsf{opt},
\end{aligned}$$

where the last inequality uses that $\vec{f}$ is a $(1+\epsilon)$-approximate transshipment solution. □

7.8.1 Parallelizing the Expected SSSP Algorithm

In this section, we parallelize the algorithm `ESSSP` by removing its sequential recursive calls which would, if left unchecked, blow up the parallel running time of the algorithm. This is because every time `ESSSP` calls itself, it requires a transshipment flow of a new recursive graph, which in turn requires an ℓ_1-embedding of it.

Our solution is to avoid computing an ℓ_1-embedding (from scratch) at each step of the recursion. With an ℓ_1-embedding at hand for a given recursive graph, we can compute the transshipment flow without any recursion by invoking Corollary 7.3.3. Naively, one might hope that a subset of the original vectors of the ℓ_1-embedding automatically produces an ℓ_1-embedding for the recursive graph instance. This is not true in general, but we can *force* it to happen at a cost: we add a set of *virtual* edges to the recursive graph instance (which may change its metric) that come from a *spanner* of the ℓ_1-metric on a subset of the original vectors. These edges may not exist in the original graph G, so the shortest path tree T of that instance may include edges that do not correspond to edges in the original graph. Unraveling the recursion, the final tree T may also contain virtual edges not originally in G. However, we ensure that these virtual edges do not change the metric of the *original* graph G. Hence, while the tree T is not a *spanning* SSSP tree, it still fulfills its purpose when it is called in the algorithm `TS-to-SSSP` of Section 7.9. (If a *spanning* SSSP tree is explicitly required for the final output of the original graph instance, then `ESSSP` can be called in `TS-to-SSSP` once on the initial recursion loop, and `ESSSP-Rec` called on every recursive instance of `TS-to-SSSP` afterwards.)

Lemma 7.8.14

Let $G=(V,E)$ be a connected graph with n vertices and m edges with aspect ratio poly(n), let $\epsilon>0$ be a parameter. Given graph G, a source $s\in V$, and an ℓ_1-embedding of it into $O(\log n)$ dimensions with distortion polylog(n), we can compute, in $\tilde{O}(m)$ work and polylog(n) time, a set E^+ of *virtual* edges supported on V and a tree $T\subseteq E\cup E^+$ such that

1. For all edges $(u,v)\in E^+$, $w(u,v)\geq d_G(u,V)$, i.e., the edges E^+ do not change the metric of G
2. T is a $(1+\epsilon)$-approximate expected s-SSSP tree on $G\cup E^+$

We now present the recursive algorithm that proves Lemma 7.8.14. Let the original graph be G_0 and the current recursive instance be G, and let n and m always indicate the number of vertices and edges of the original graph G_0. We highlight in blue all modifications of `ESSSP` that we make.

One notable difference is that we always enforce the recursive vertex set V' to be a subset of the current vertex set V. This allows us to compare the metric of the new graph G' to

Algorithm 13 `ESSSP-Rec`$(G = (V, E), \{y_v \in \mathbb{R}^k : v \in V\}, s, b, (1 + 3\epsilon)\alpha)$

```
Global variables: $G_0 = (V_0, E_0)$ is the original input graph; $\{x_v \in \mathbb{R}^{O(\log n)} : v \in V\}$ is an initial $\ell_1$-embedding of $G_0$ with distortion $D = \text{polylog}(n)$ given as input
Assumption: $k = O(\log^2 n)$ and $\{y_v\}$ satisfies $\frac{1}{8kD} d_G(u, v) \le \mathbb{1} y_u - y_v \le (D + 6) \cdot d_G(u, v)$ for all $u, v \in V$; demand vector $b$ satisfies $b_s > 0$ and $b_t \le 0$ for all $t \in V \setminus s$
 1: $\{8kD \cdot y_v\}$ is an $\ell_1$-embedding of $G$ with distortion $8kD(D + 6) = \text{polylog}(n)$. Apply dimension reduction to obtain an $\ell_1$-embedding of $G$ into $O(\log n)$ dimensions instead of $O(\log^2 n)$, with a slightly worse but still polylog$(n)$ distortion (see proof of Theorem 7.4.6). Using it and Corollary 7.3.3, compute a $(1 + \epsilon)$-approximate transshipment $f$ on $G$ with demand vector $b$
 2: Initialize the digraph $\overrightarrow{A} \leftarrow \emptyset$
 3: Every vertex $u \in V \setminus s$ with $\text{in}(u) \ne \emptyset$ independently samples a random neighbor $v \in \text{out}(u)$ with probability $\vec{f}(u, v)/\vec{f}_{\text{out}}(u)$ and adds arc $(u, v)$ to $\overrightarrow{A}$
 4: Add a self-loop $(s, s)$ of zero weight to $\overrightarrow{A}$
 5: Let $A$ be the undirected version of $\overrightarrow{A}$
 6: Initialize $G' \leftarrow (\emptyset, \emptyset)$ as an empty undirected graph        ▷ Graph to be recursed on, with $\le n/2$ vertices
 7: Initialize $b'$ as an empty vector        ▷ Demands to be recursed on
 8: for each connected component $C$ of $A$ do
 9:     $c(C) \leftarrow$ total weight of edges in the (unique) cycle in $C$ (possibly the self-loop $(s, s)$)
10:     Let $T_C$ be the graph $C$ with its (unique) cycle contracted into a single vertex $r_C$    ▷ $T_C$ is a tree
11:     Let $v_C \in V$ be an arbitrary vertex on the cycle in $C$
12:     Add the vertex $v_C$ to $G'$, and set demand $b'_{v_C} \leftarrow \sum_{v \in V(C)} b_v$
13: for each edge $(u, u')$ in $E$ do
14:     Let $C$ and $C'$ be the connected components of $A$ containing $u$ and $u'$, respectively
15:     if $C \ne C'$ then
16:         Add an edge between $v_C$ and $v_{C'}$ with weight $w(u, u') + d_{T_C}(u, r_C) + d_{T_{C'}}(u', r_{C'}) + c(C) + c(C')$
17: Note that $s = v_{C_s}$, where $C_s$ is the component of $A$ containing $s$
18: Collapse parallel edges of $G'$ by only keeping the parallel edge with the smallest weight
19: Call Lemma 7.8.16 on the original $\ell_1$-embedding vectors $\{x_v : v \in V'\}$, returning a set of vectors $\{y'_v \in \mathbb{R}^{O(\log^2 n)} : v \in V'\}$ that satisfy (7.25)
20: Call Lemma 7.8.17 on $\{y'_v : v \in V'\}$, returning a set $E^+$ of edges supported on $V'$ that satisfy both conditions of Lemma 7.8.17
21: Recursively call ESSSP-Rec$(G' \cup E^+, \{y'_v : v \in V'\}, s, b', \alpha)$, obtaining an $\alpha$-approximate expected SSSP tree $T'$ of $G' \cup E^+$
22: Initialize $T \leftarrow \emptyset$        ▷ The expected SSSP tree
23: for each edge $(v, v')$ in $T' \cap E$ do
24:     Let $(u, u') \in E$ be the edge responsible for adding edge $(v, v')$ to $G'$
25:     Add edge $(u, u')$ to $T$        ▷ Continued on next page
```

Algorithm 13 `ESSSP-Rec`$(G = (V, E), s, b, (1 + 3\epsilon)\alpha)$, continued

```
26: for each edge $(v, v')$ in $T' \setminus E$ do
27:     Let $(u, u') \in E$ be the edge responsible for adding edge $(v, v')$ to $G'$
28:     Let $C$ and $C'$ be the connected components of $A$ containing $u$ and $u'$, respectively
29:     Add edge $(v, v')$ to $T$ with weight $w(v, v') - c(C) - c(C')$   ▷ These edges are not in
        the original edge set $E_0$
30: for each connected component $C$ of $A$ do
31:     Remove an arbitrary edge from the (unique) cycle inside $C$, and add the resulting
        tree to $T$
32: return $T$
```

the current graph G in a more direct way.

Claim 7.8.15

Each edge $(v_C, v_{C'})$ added on line 16 has weight at least $d_G(v_C, v_{C'})$.

Proof. Let $(u, u') \in E$ be the edge responsible for adding the edge $(v_C, v_{C'})$. We construct a path in G from v_C to $v_{C'}$ whose distance is at most $w(u, u') + d_{T_C}(u, r_C) + d_{T_{C'}}(u', r_{C'}) + c(C) + c(C')$. By construction of component C, there is a path inside C from v_C to u of length at most $c(C) + d_{T_C}(u, r_C)$. Similarly, there is a path inside C' from u' to $v_{C'}$ of length at most $c(C') + d_{T_{C'}}(u', r_{C'})$. Adding the edge (u, u') completes the path from v_C to $v_{C'}$. □

Lemma 7.8.16

Given the vectors $\{x_v\}$, we can compute, in $\tilde{O}(m)$ work and polylog(n) time, a set of vectors $\{y_v \in \mathbb{R}^{O(\log^2 n)} : v \in V'\}$ such that w.h.p., for all vertices $v_C, v_{C'} \in V'$,

$$\mathbb{1}x_{v_C} - x_{v_{C'}} + 2(c(C) + c(C')) \le \mathbb{1}y_{v_C} - y_{v_{C'}} \le \mathbb{1}x_{v_C} - x_{v_{C'}} + 6(c(C) + c(C')). \tag{7.25}$$

Proof. We introduce $O(\log^2 n)$ new coordinates, indexed by $(i, j) \in \mathbb{Z} \times [s]$. For a given component C of A at this instance, initialize $y_{v_C} = x_{v_C}$ without the new coordinates. Let $i \in \mathbb{Z}$ be the integer i such that $c(C) \in [2^i, 2^{i+1})$, which can take one of $O(\log n)$ values in $\mathbb{Z}$. For each $j \in [s]$, let the coordinate at index (i, j) take value $\pm 5 \cdot 2^i / s$, each with probability $1/2$. This concludes the construction of y_{v_C}.

Fix vertices $v_C, v_{C'} \in V'$. Let i, i' be the integers such that $c(C) \in [2^i, 2^{i+1})$ and $c(C') \in [2^{i'}, 2^{i'+1})$. If $i \neq i'$, then by construction, the ℓ_1 distance incurred by the additional $O(\log^2 n)$ coordinates is exactly $5 \cdot 2^i + 5 \cdot 2^{i'}$. Moreover, by the input guarantee of $\{x_v : v \in V\}$,

$$d_{G_0}(v_C, v_{C'}) \le \mathbb{1}x_{v_C} - x_{v_{C'}} \le D \cdot d_G(v_C, v_{C'}).$$

Therefore,

$$\begin{aligned}
\mathbb{1}x_{v_C} - x_{v_{C'}} + 2\cdot(c(C)+c(C')) &\le \mathbb{1}x_{v_C} - x_{v_{C'}} + 2\cdot 2^{i+1} + 2\cdot 2^{i'+1} \\
&\le \mathbb{1}x_{v_C} - x_{v_{C'}} + 5\cdot 2^{i} + 5\cdot 2^{i'} \\
&= \mathbb{1}y_{v_C} - y_{v_{C'}} \qquad (7.26) \\
&\le \mathbb{1}x_{v_C} - x_{v_{C'}} + 6\cdot 2^{i} + 6\cdot 2^{i'} \\
&\le \mathbb{1}x_{v_C} - x_{v_{C'}} + 6\cdot(c(C)+c(C')),
\end{aligned}$$

as promised. Now suppose that $i = i'$. Then, for each of the s coordinates (i, j), there is a $1/2$ probability of contributing 0 to $\mathbb{1}y_{v_C} - y_{v_{C'}}$, and a $1/2$ probability of contributing $6\cdot 2^i/s$. Since $s = O(\log n)$, by a simple Chernoff bound, the probability that the total contribution of these s coordinates is in $[4\cdot 2^i/s, 6\cdot 2^i/s]$ is at least $1 - 1/\text{poly}(n)$. A similar bound to (7.26) proves the claim. $\square$

Lemma 7.8.17

Given vectors $\{y_v \in \mathbb{R}^{O(\log^2 n)}\}$ satisfying (7.25), we can compute, in $\tilde{O}(|E|)$ work and polylog(n) time, a set E^+ of edges supported on $V' \subseteq V$ such that

1. For all edges $(v_C, v_{C'}) \in E^+$, $w_{E^+}(v_C, v_{C'}) \ge d_{G_0}(v_C, v_{C'}) + 2(c(C)+c(C'))$,
2. For all vertices $u, v \in V'$,

$$\frac{1}{8kD}\cdot d_{G'\cup E^+}(u,v) \le \mathbb{1}y_u - y_v \le (D+6)\cdot d_{G'\cup E^+}(u,v).$$

Proof. Let H be an $(8kD)$-spanner of the (complete) ℓ_1-metric induced by the vectors $\{y_v : v \in V'\}$. That is, for all $u, v \in V'$,

$$\mathbb{1}y_u - y_v \le d_H(u,v) \le 8kD\cdot \mathbb{1}y_u - y_v.$$

We show later how to compute it efficiently, but for now, assume we have computed such a graph H. We set the edges E^+ as simply the edges of H (with the same weights).

Every edge $(v_C, v_{C'}) \in E^+$ satisfies

$$w_{E^+}(v_C, v_{C'}) \ge d_H(v_C, v_{C'}) \ge \mathbb{1}y_{v_C} - y_{v_{C'}} \overset{(7.26)}{\ge} \mathbb{1}x_{v_C} - x_{v_{C'}} + 2(c(C)+c(C')),$$

and $\mathbb{1}x_{v_C} - x_{v_{C'}} \ge d_{G_0}(v_C, v_{C'})$ since $\{x_v\}$ is an ℓ_1-embedding for G_0, fulfilling condition (1). For each pair of vertices $u, v \in V'$,

$$d_{G'\cup E^+}(u,v) \le d_H(u,v) \le 8kD\cdot \mathbb{1}y_u - y_v,$$

fulfilling the lower bound in condition (2).

For the upper bound in condition (2), it suffices to show that, for all edges $(v_C, v_{C'}) \in E' \cup E^+$,

$$\mathbb{1}y_u - y_v \leq (D+6) \cdot w_{G' \cup E^+}(u,v).$$

If $(v_C, v_{C'}) \in E^+$, then

$$\mathbb{1}y_{v_C} - y_{v_{C'}} \leq d_H(v_C, v_{C'}) \leq w_H(v_C, v_{C'}) = w_{G' \cup E^+}(v_C, v_{C'}).$$

Otherwise, if $(v_C, v_{C'}) \in E'$, then for the edge $(u, u') \in E$ with $u \in C$ and $u' \in C'$ responsible for this edge,

$$\begin{aligned}
\mathbb{1}y_{v_C} - y_{v_{C'}} &\overset{(7.26)}{\leq} \mathbb{1}x_{v_C} - x_{v_{C'}} + 6(c(C) + c(C')) \\
&\leq D \cdot d_{G_0}(v_C, v_{C'}) + 6w_{G'}(v_C, v_{C'}) \\
&\overset{(7.27)}{\leq} D \cdot d_G(v_C, v_{C'}) + 6w_{G'}(v_C, v_{C'}) \\
&\overset{\text{Clm.7.8.15}}{\leq} D \cdot w_{G'}(v_C, v_{C'}) + 6w_{G'}(v_C, v_{C'}),
\end{aligned}$$

assuming the following inequality holds:

$$d_G(u,v) \geq d_{G_0}(u,v) \qquad \forall u, v \in V. \tag{7.27}$$

To prove (7.27), we assume by induction (on the recursive structure of `ESSSP-Rec`) that $d_G(u,v) \geq d_{G_0}(u,v)$ for all $u, v \in V$. By Claim 7.8.15, the edges (u,v) added to G' on line 16 have weight at least $d_G(u,v) \geq d_{G_0}(u,v)$. Moreover, since condition (1) of the lemma is satisfied, we have $w(u,v) \geq d_{G_0}(u,v)$ for all edges $(u,v) \in E^+$ as well. It follows that $d_{G_0}(u,v) \leq d_{G' \cup E^+}(u,v)$ for all $u, v \in V'$, fulfilling the inequality as well as completing the induction. This concludes the upper bound of condition (2), as well as the proof of Lemma 7.8.17 modulo the construction of the spanner H.

Finally, we show how to compute the spanner H of the ℓ_1-metric induced by the vectors $\{y_v \in \mathbb{R}^k : v \in V\}$. We use a randomly shifted grid approach similar to the one in Algorithm 9. Assume without loss of generality (by scaling and rounding) that all coordinates are positive integers with magnitude at most $M = \text{poly}(n)$. We repeat the following process $O(\log n)$ times, computing a graph H_j on trial j. For each W a power of two in the range $[1, 8kM]$, choose independent, uniformly random real numbers $r_1, \ldots, r_k \in [0, W)$, and declare equivalence classes on the vertices where two vertices $u, v \in V$ are in the same class if $\lfloor (y_u)_i + r_i \rfloor_W = \lfloor (y_v)_i + r_i \rfloor_W$ for all coordinates $i \in [k]$. For each equivalence class $U \subseteq V$, select an arbitrary vertex $u \in U$ as its representative, and for all $v \in U \setminus \{u\}$, add an edge (u,v) to H_j of weight $\mathbb{1}y_u - y_v$. This concludes the construction of graph H_j, which has $O(|V| \log |V|)$ edges, since each value of W adds at most $|V|$ edges. The spanner H is the union of all graphs H_j and has size $O(|V| \log |V| \log n)$.

Since all added edges (u,v) have weight $\mathbb{1}y_u - y_v \geq d_G(u,v)$, distances in the final spanner

H are at least the distances in G. To show that distances in H are not stretched too far, we show that for any vertices $u, v \in V$, with constant probability, each graph H_j satisfies $d_{H_j}(u,v) \le 8kD \cdot d_G(u,v)$. Then, since there are $\Theta(\log n)$ graphs H_j, the probability that we do not have $d_H(u,v) \le d_{H_j}(u,v) \le 8kD \cdot d_G(u,v)$ for any j is $1/\text{poly}(n)$, as promised.

Let W be the smallest power of two greater than or equal to $2\mathbb{1}y_u - y_v$. Note that $\mathbb{1}y_u - y_v \le 2kM$, so such a W always exists. It is not hard to show that, for each coordinate $i \in [k]$, the probability that $\lfloor (y_u)_i + r_i \rfloor_W = \lfloor (y_v)_i + r_i \rfloor_W$ is exactly $1 - |(y_u - y_v)_i|/W \ge 1/2$. Therefore, the probability that $\lfloor (y_u)_i + r_i \rfloor_W = \lfloor (y_v)_i + r_i \rfloor_W$ for all i is

$$\prod_{i=1}^{k} \left(1 - \frac{|(y_u - y_v)_i|}{W}\right) \ge \prod_{i=1}^{k} \exp\left(-2\frac{|(y_u - y_v)_i|}{W}\right) = \exp\left(-2\sum_{i=1}^{k} \frac{|(y_u - y_v)_i|}{W}\right)$$
$$= \exp\left(-2\frac{\mathbb{1}y_u - y_v}{W}\right) \ge \exp(-1),$$

which is a constant. Therefore, with at least constant probability, u and v belong to the same equivalence class for W. In this case, since all vertices in this equivalence class have their vectors in a cube of side length W, we either added the edge (u,v) of weight $\mathbb{1}y_u - y_v \le D \cdot d_G(u,v)$, or we selected some vertex v' and added the edges (u,v') and (v,v') of total weight at most

$$\mathbb{1}y_u - y_{v'} + \mathbb{1}y_v - y_{v'} \le 2kW \le 2k \cdot 4\mathbb{1}y_u - y_v \le 2k \cdot 4 \cdot D \cdot d_G(u,v).$$

Thus, H is a $8kD$-spanner w.h.p. ☐

We now argue that, for the input graph G_0, source $s \in V_0$, demand vector b_0, and an ℓ_1-embedding $\{x_v \in \mathbb{R}^{O(\log n)} : v \in V_0\}$ of G_0 with distortion D, $\texttt{ESSSP-Rec}(G_0, \{x_v\}, s, b_0, (1+3\epsilon)^{\log_2 n})$ returns a $(1+3\epsilon)^{\log_2 n}$-approximate expected s-SSSP tree. We will follow the arguments from Section 7.8 almost line-by-line; the only changes we will highlight in blue. Define $V^*, b^+, \overrightarrow{C}, \vec{f}$ on the input graph $G = (V, E)$ and demands b identically as in Section 7.8. As before, define $\mathsf{opt} = \mathsf{opt}_G(b)$.

Claim 7.8.18: Restatement of Claim 7.8.12

Define T' and T as in $\texttt{ESSSP-Rec}$. For each component C of A and each vertex v in C, we have $d_T(s,v) \le d_{T'}(s', v_C) + d_{T_C}(t, r_C)$.

Proof. Follows easily by construction (lines 23 to 31), so proof is omitted. Observe that the extra terms $-c(C) - c(C')$ in line 29 are necessary here. ☐

Corollary 7.8.19: Restatement of Corollary 7.8.11

Over the randomness of `ESSSP-Rec` (in particular, the random choices on line 3),

$$\mathbb{E}\left[\sum_{t\in V^*} b^+(t)\cdot d_{T_C}(u,r_C)+\mathsf{opt}_{G'\cup E^+}(b')\right]\le \mathbb{1}\overrightarrow{C}\vec{f}+2\epsilon\cdot\mathsf{opt}.$$

The optimum can only decrease with the addition of edges E^+ to G', so we have

$$\mathsf{opt}_{G'\cup E^+}(b')\le\mathsf{opt}_{G'}(b'). \tag{7.28}$$

Claim 7.8.20: Restatement of Claim 7.8.13

Over the entire randomness of `ESSSP-Rec` (including the recursion at line 21), we have

$$\mathbb{E}\left[\sum_{t\in V} b^+(t)\cdot d_T(s,t)\right]\le\alpha(1+3\epsilon)\mathsf{opt}.$$

Proof. We follow the proof of Claim 7.8.13. By Claim 7.8.3, we have $b^+(t)=0$ for all $t\in V\setminus V^*$, so it suffices to prove

$$\mathbb{E}\left[\sum_{t\in V^*} b^+(t)\cdot d_T(s,t)\right]\le\alpha(1+3\epsilon)\mathsf{opt}.$$

By recursion, we have the guarantee

$$\mathbb{E}\left[\sum_{C:v_C\ne s'} b'_{v_C}\cdot d_{T'}(s',v_C)\right]\le\alpha\mathsf{opt}.$$

For each vertex t, let C_t be the component of A containing t. We have

$$\begin{aligned}\mathbb{E}\left[\sum_{t\in V^*} b^+(t)\cdot d_T(s,v)\right] &\overset{\text{Clm. 7.8.18}}{\le}\mathbb{E}\left[\sum_{t\in V^*}\left(b^+(t)\cdot d_{T'}(s',v_{C_t})+d_{T_{C_t}}(t,r_{C_t})\right)\right]\\ &=\mathbb{E}\left[\sum_{t\in V^*} b^+(t)\cdot d_{T'}(s',v_{C_t})\right]+\mathbb{E}\left[\sum_{t\in V^*} b^+(t)\cdot d_{T_{C_t}}(t,r_{C_t})\right]\\ &=\mathbb{E}\left[\sum_{C:v_C\ne s'} b'_{v_C}\cdot d_{T'}(s',v_C)\right]+\mathbb{E}\left[\sum_{t\in V^*} b^+(t)\cdot d_{T_{C_t}}(t,r_{C_t})\right]\end{aligned}$$

$$
\begin{aligned}
&\leq \alpha \mathbb{E}[\mathsf{opt}_{G' \cup E^+}(b')] + \mathbb{E}\left[\sum_{t \in V^*} b^+(t) \cdot d_{T_{C_t}}(t, r_{C_t})\right] \\
&\overset{(7.28)}{\leq} \alpha \mathbb{E}[\mathsf{opt}_{G'}(b')] + \mathbb{E}\left[\sum_{t \in V^*} b^+(t) \cdot d_{T_{C_t}}(t, r_{C_t})\right] \\
&\leq \alpha \left(\mathbb{E}[\mathsf{opt}_{G'}(b')] + \mathbb{E}\left[\sum_{t \in V^*} b^+(t) \cdot d_{T_{C_t}}(t, r_{C_t})\right]\right) \\
&\overset{\text{Cor. 7.8.19}}{\leq} \alpha \left(\mathbb{1}\overrightarrow{C}\vec{f} + 2\epsilon \cdot \mathsf{opt}\right) \\
&\leq \alpha((1+\epsilon)\mathsf{opt} + 2\epsilon \cdot \mathsf{opt}) \\
&= \alpha(1+3\epsilon)\mathsf{opt},
\end{aligned}
$$

where the last inequality uses that $\vec{f}$ is a $(1+\epsilon)$-approximate transshipment solution. □

Thus, by unraveling the recursion and repeatedly applying Claim 7.8.20, the algorithm `ESSSP-Rec` computes a tree T for G_0, possibly with virtual edges not in G_0, such that

$$
\mathbb{E}\left[\sum_{t \in V} b^+(t) \cdot d_T(s,t)\right] \leq (1+3\epsilon)^{\log_2 n} \mathsf{opt}.
$$

It remains to show that T is a "valid" approximate expected SSSP tree in the following sense: the virtual edges added to T do not change the metric on G_0. This property is sufficient when the expected SSSP algorithm is used in `TS-to-SSSP`.

For each vertex $v \in V_0$, consider the recursive instances with $v \in V$; let $C_i(v)$ be the components containing v on previous levels of recursion, with C_i as the value at recursion level i. We will need the following quick claim:

Claim 7.8.21

Consider a recursion level i with input graph $G = (V, E)$. For all vertices $v \in V$, the lengths $c(C_j(v))$ of the cycles satisfy $c(C_{j+1}(v)) \geq 2c(C_j(v))$ for all $j < i$.

Proof. If $v = s$, then it holds because $c(C_j(v)) = 0$ for all j, so assume that $v \neq s$. For a fixed recursion level j, by construction, every edge adjacent to v has weight at least $c(C_i(v))$ in the graph G' at that level. The cycle $C_{i+1}(v)$ must contain at least two such edges, so its total length is at least $2c(C_i(v))$. □

For each virtual edge (u, v) in T, there exists some recursion level i and some edge $(u, v) \in E^+$ at that level of weight

$$
w_{E^+}(u,v) = w_T(u,v) + \sum_{j=0}^{i} (c(C_j(u)) + c(C_j(v))).
$$

By Claim 7.8.21, the sequence $c(C_0(u)), c(C_1(u)), \ldots, c(C_i(u)) = c(C)$ has each term at least double the previous, which means that two times the last term is at least the sum of all terms:

$$2c(C) = 2c(C_j(u)) \geq \sum_{j=0}^{i} c(C_j(u)).$$

Similarly,

$$2c(C') = 2c(C_j(v)) \geq \sum_{j=0}^{i} c(C_j(v)).$$

By condition (1) of Lemma 7.8.17,

$$w_{E^+}(u,v) \geq d_{G_0}(u,v) + 2(c(C) + c(C')).$$

Combining these inequalities gives

$$w_T(u,v) = w_{E^+}(u,v) - \sum_{j=0}^{i}(c(C_j(u)) + c(C_j(v))) \geq w_{E^+}(u,v) - 2(c(C) + c(C')) \geq d_{G_0}(u,v),$$

as claimed.

7.9 Sampling a Primal Tree

In this section, we prove Theorem 7.3.7. by reducing the problems of computing a $(1+\epsilon)$-approximate SSSP tree and potential to the approximate transshipment problem, and then using the expected SSSP tree subroutine `ESSSP-Rec` of Section 7.8.1. Most of these ideas originate from [14], and we adapt their ideas and present them here for completeness. In particular, we do not claim any novelty in this section.

Theorem: Restatement of Theorem 7.3.7

Let $G = (V, E)$ be a graph with n vertices and m edges, and let $\epsilon > 0$ be a parameter. Given graph G, a source $s \in V$, and an ℓ_1-embedding of it into $O(\log n)$ dimensions with distortion polylog(n), we can compute a $(1+\epsilon)$-approximate SSSP tree and potential in additional $\tilde{O}(m/\epsilon^2)$ work and $\tilde{O}(1/\epsilon^2)$ time.

We now briefly describe our algorithm for Theorem 7.3.7. First, we run the expected SSSP algorithm `ESSSP-Rec` with demands $\sum_{v \neq s}(\mathbb{1}_v - \mathbb{1}_s)$, obtaining distances that are near-optimal *in expectation* in the two different ways. Of course, what we need is that *all* distances are near-optimal. This is where the potential ϕ is useful: using it, we can approximately tell which vertices have near-optimal distances. Then, among the vertices $V' \subseteq V \setminus s$ whose distances are not near-optimal, we then compute another transshipment instance with demands $\sum_{v\in V'}(\mathbb{1}_v - \mathbb{1}_s)$ and repeat the process. As long as the set of remaining vertices V' drops by a constant factor each round in expectation, we only require $O(\log n)$ rounds w.h.p.

To construct the potential, our strategy is simple: we simply take the coordinate-wise maximum of all potentials ϕ found over the iterations (assuming $\phi(s) = 0$ always). For each vertex $v \in V \setminus s$, since at least one iteration computes a near-optimal distance for v, the corresponding potential is also near-optimal.

Constructing the specific SSSP tree requires a little more care. We now describe our algorithm in pseudocode below.

Algorithm 14 `TS-to-SSSP`$(G = (V, E), \beta \in (0, 1], \{x_v \in \mathbb{R}^{O(\log n)} : v \in V\})$

```
Assumption: $\{x_v \in \mathbb{R}^{O(\log n)} : v \in V\}$ is an $\ell_1$-embedding of $G$ with distortion polylog$(n)$
1: Initialize $V' \leftarrow V \setminus s$ ▷ $V'$ is the set of vertices whose distances still need to be computed
2: Initialize $d^* : V \setminus s \to \mathbb{R} \cup \infty$ as $d^*(v) = \infty$ everywhere    ▷ $d^*$ tracks the best distance found for each vertex $v$
3: Initialize $p^* : V \setminus s \to V \cup \{\perp\}$ as $p^*(v) = \perp$ everywhere    ▷ $p^*$ is the "parent" function, used to construct the final SSSP tree
4: Initialize $\phi^* : V \setminus s \to \mathbb{R}$ as $\phi^*(v) = 0$ everywhere    ▷ $\phi^*$ tracks the best potential found for each vertex $v$
5: while $V' \neq \emptyset$ do
6:     With the $\ell_1$-embedding $\{x_v : v \in V\}$, call Corollary 7.3.3 on demands $\sum_v(\mathbb{1}_v - \mathbb{1}_s)$ to obtain $(1 + \epsilon/10)$-approximate flow-potential pair $(f, \phi)$, where $f$ is an acyclic flow and $\phi(s) = 0$
7:     Set demands $b \leftarrow \sum_{v\in V'}(\mathbb{1}_v - \mathbb{1}_s)$ for this iteration
8:     With the $\ell_1$-embedding $\{x_v : v \in V\}$, call ESSSP-Rec$(G, \{x_v : v \in V\}, s, b, \Theta(\frac{\epsilon}{\log n}))$ to obtain a SSSP tree $T$ with $\mathbb{E}\left[\sum_{v\in V'} d_T(s, v)\right] \le (1 + \frac{\epsilon}{10})\mathsf{opt}$ ▷ Satisfies the recursion in (7.13)
9:     Compute distances $d_T(s, v)$ for all $v \in V'$
10:    for each vertex $v \in V'$ in parallel do
11:        Let $p(v)$ be the second-to-last vertex on the path from $s$ to $v$ in $T$
12:        if $d_T(s, v) < d^*(v)$ then
13:            Update $d^*(v) \leftarrow d_T(s, v)$
14:            Update $p^*(v) \leftarrow p(v)$
15:        Update $\phi^*(v) \leftarrow \max\{\phi^*(v), \phi(v)\}$
16:        if $d_T(s, v) \le (1 + \epsilon)\phi_v$ then
17:            Remove $v$ from $V'$
18: Initialize $T^* \leftarrow \emptyset$    ▷ $T^*$ is the final SSSP tree that we compute
19: for each vertex $v \in V \setminus s$ do
20:     Add edge $(v, p^*(v))$ to $T^*$
21: return SSSP tree $T^*$ and potential $\phi^*$ (augmented with $\phi^*(s) := 0$)
```

Claim 7.9.2

If TS-to-SSSP finishes, then the potential ϕ^* that it returns is a $(1+\epsilon)$-approximate s-SSSP potential, and the returned T^* is a $(1+\epsilon)$-approximate SSSP tree.

Proof. TS-to-SSSP essentially takes the coordinate-wise maximum over all potentials ϕ that Algorithm $\mathcal{A}$ computes over the iterations. For each vertex $v \in V \setminus s$, since it was removed from V' at some point, some potential ϕ computed satisfies $(1+\epsilon)\phi_v \ge d_T(s,v) \ge d(s,v)$ (line 16), so the final potential ϕ^* also satisfies $(1+\epsilon)\phi^*_v \ge d(s,v)$. Since each potential ϕ satisfies $|\phi(u) - \phi(v)| \le w(u,v)$ or each edge (u,v), by Observation 7.2.11, so does the coordinate-wise maximum ϕ^*. Therefore, ϕ^* is an SSSP potential.

Next, we show that the graph T^* returned is indeed a tree. Observe the following invariant: for each vertex $v \in V$, whenever $p^*(v) \neq \perp$, we have $d^*(p^*(v)) < d^*(v)$. This is because whenever $d^*(v)$ is updated to $d_T(s,v)$ on line 13, we must have $d_T(s,p^*(v)) < d_T(s,v)$ for that tree T, so $d^*(p^*(v))$ would be updated as well if it was still at least $d_T(s,v)$. Therefore, the edges $(v,p^*(v))$ at the end of the **while** loop must also satisfy $d^*(p^*(v)) < d^*(v)$, so the edges are acyclic. Since there are $n-1$ edges total, T^* is a tree. Finally, to show that T^* is a $(1+\epsilon)$-approximate SSSP tree, we show that for each vertex $v \in V$, $d_{T^*}(s,v) \le d^*(v)$. To see that this is sufficient, observe that since every vertex $v \in V \setminus s$ was removed from V', we have $d_T(s,v) \le (1+\epsilon)\phi_v \le (1+\epsilon)d(s,v)$ at some point (line 16), so on this iteration, $d^*(v)$ would be updated to at most $(1+\epsilon)d(s,v)$.

We prove by induction on the ordering of $d^*(v)$ (from smallest to largest) that $d_{T^*}(s,v) \le d^*(v)$. If $p^*(v) = s$, then since $d^*(v)$ and $p^*(v)$ are updated at the same time (lines 13 and 14), the value $d^*(s) = d_T(s,v) = w(s,v)$ cannot be changed after $p^*(v)$ was set to s. Therefore, $d_{T^*}(s,v) = d^*(v)$. Otherwise, suppose that $p^*(v) = u \neq s$. Let T be the tree computed on the iteration when $p^*(v)$ was updated to its final value. We have

$$d_{T^*}(s,v) = d_{T^*}(s,u) + w(u,v) \overset{\text{(ind.)}}{\le} d^*(u) + w(u,v) \le d_T(s,u) + w(u,v),$$

where the (ind.) stands for applying the inductive statement on vertex u. This completes the induction and the claim. □

For the remainder of this section, we show that the **while** loop runs for only $O(\log n)$ iterations w.h.p. Consider the following potential function $\sum_{v \in V'} b_v d(s,v)$; we will show that it drops by a constant factor in expectation on each iteration of the **while** loop. Since the graph G has polynomial aspect ratio, the potential function can only decrease by a constant factor $O(\log n)$ times, so the lemma below suffices to finish Theorem 7.3.7.

Lemma 7.9.3

On each iteration of the **while** loop (line 5), the quantity $\sum_{v \in V'} d(s,v)$ drops by a constant factor in expectation.

Proof. Define $\overline{d}(v) := \mathbb{E}[d_T(s,v)]$ over the randomness of ESSSP on this iteration, which satisfies $\sum_{v\in V'} \overline{d}(v) \le (1+\epsilon/10)\sum_{v\in V'} d(s,v)$ since T is an $(1+\epsilon/10)$-approximate expected SSSP tree with demands $\sum_{v\in V'}(\mathbb{1}_v - \mathbb{1}_s)$. Since (f,ϕ) is a $(1+\epsilon/10)$-approximate flow-potential pair, we have $\mathbb{1}f \le (1+\epsilon/10)\sum_{v\in V} b_v\phi_v$. We have, for small enough ϵ,

$$\begin{aligned}\sum_{v\in V'} \overline{d}(v) &\le \left(1+\frac{\epsilon}{10}\right)\sum_{v\in V'} d(s,v)\\ &\le \left(1+\frac{\epsilon}{10}\right)\mathsf{opt}\left(\sum_{v\in V'}(\mathbb{1}_v - \mathbb{1}_s)\right)\\ &\le \left(1+\frac{\epsilon}{10}\right)\mathbb{1}f\\ &\le \left(1+\frac{\epsilon}{10}\right)^2\sum_{v\in V} b_v\phi_v \le \left(1+\frac{\epsilon}{4}\right)\sum_{v\in V} b_v\phi_v = \left(1+\frac{\epsilon}{4}\right)\sum_{v\in V'}\phi_v,\end{aligned}$$

which implies that

$$\sum_{v\in V'}(\overline{d}(v)-\phi_v) \le \frac{\epsilon}{4}\sum_{v\in V'}\phi_v.$$

Observe that for all $v\in V'$, $\overline{d}(v) \ge d(s,v) \ge \phi_v - \phi_s = \phi_v$, so $\overline{d}(v)-\phi_v \ge 0$. Let $V'_{\text{good}} \subseteq V'$ be the vertices $v\in V'$ with

$$\overline{d}(v)-\phi_v \le \frac{\epsilon}{2}\phi_v.$$

By a Markov's inequality-like argument, we have

$$\sum_{v\in V'_{\text{good}}}\phi_v \ge \frac{1}{2}\sum_{v\in V'}\phi_v; \tag{7.29}$$

otherwise, we would have

$$\sum_{v\in V'}(d(v)-\phi_v) \ge \sum_{v\in V'\setminus V'_{\text{good}}}(d(v)-\phi_v) \ge \frac{\epsilon}{2}\sum_{v\in V'\setminus V'_{\text{good}}}\phi_v > \frac{\epsilon}{4}\sum_{v\in V'}\phi_v,$$

a contradiction.

For each $v\in V'_{\text{good}}$, by Markov's inequality on the nonnegative random variable $d_T(s,v)-\phi_v$ (which has expectation $\overline{d}(v)-\phi_v \le \frac{\epsilon}{2}\phi_v$), with probability at least 1/2, we have

$$d_T(s,v)-\phi_v \le \epsilon\,\phi_v \iff d_T(s,v) \le (1+\epsilon)\phi_v,$$

so vertex v is removed from V' with probability at least 1/2. In other words, the contribution of v to the expected decrease of $\sum_{u\in V'} d(s,u)$ is at least $\frac{1}{2}\phi_v$. Since

$$d(s,v) \le d_T(s,v) \le (1+\epsilon)\phi_v \implies \phi_v \ge (1+\epsilon)^{-1}d(s,v),$$

this expected decrease is at least $\frac{1}{2(1+\epsilon)}\phi_v$. Summing over all $v \in V'_{\text{good}}$, the expected decrease of $\sum_{u\in V'} d(s,u)$ is at least

$$\sum_{v\in V'_{\text{good}}} \frac{1}{2(1+\epsilon)} d(s,v) \overset{(7.29)}{\geq} \frac{1}{4(1+\epsilon)} \sum_{v\in V'} d(s,v),$$

which is a constant factor. □

7.10 Omitted Proofs

7.10.1 Proof of Lemma 7.3.5

Lemma: Restatement of Lemma 7.3.5

Given a transshipment instance with graph $G = (V,E)$ with n vertices and m edges and an integer demand vector b satisfying $|b_v| \leq M$ for all $v \in V$, we can transform G into another graph $\widehat{G}$ on n vertices and at most m edges such that $\widehat{G}$ has aspect ratio at most $n^4 M$, and $\mathsf{opt}_G(b) \leq \mathsf{opt}_{\widehat{G}}(b) \leq (1+1/n^2)\,\mathsf{opt}_G(b)$. The transformation takes $\tilde{O}(m)$ work and polylog(n) time.

Proof. Suppose that the demand vector b satisfies $b_v \in \{0,1,2,\ldots,n^C\}$ for all $v \in V$, for some constant C. First, compute a minimum spanning tree T of the graph $G = (V,E)$,[7] and compute the optimal transshipment cost where the input graph is T instead, which is easily done efficiently since T is a tree. Since T is a minimum spanning tree, it is easy to see that for any vertices $u,v \in V$, we have $d_G(u,v) \leq d_T(u,v) \leq (n-1)\cdot d_G(u,v)$, i.e., the *stretch* of T is at most $(n-1)$. Define $Z := \mathsf{opt}_T(b)$ as the optimal transshipment cost on T; it follows that

$$\mathsf{opt}_G(b) \leq Z \leq (n-1)\,\mathsf{opt}_G(b). \tag{7.30}$$

To construct $\widehat{G}$, we start with G and remove all edges of weight more than Z from G, and then add Z/n^{C+5} weight to each remaining edge in the graph. Clearly, $\widehat{G}$ has aspect ratio poly(n) and satisfies $\mathsf{opt}_G(b) \leq \mathsf{opt}_{\widehat{G}}(b)$. It remains to show that

$$\mathsf{opt}_{\widehat{G}}(b) \leq \left(1 + \frac{1}{n^2}\right)\mathsf{opt}_G(b). \tag{7.31}$$

The transshipment problem can be formulated as an uncapacitated minimum cost flow problem. It is well-known that if the demands of a minimum cost flow problem are integral, then there exists an optimal flow that is integral. Let f be this integral flow for demands

[7] This can be computed work-efficiently in Parallel, e.g., with Boruvka's algorithm.

b. Then, f cannot carry any flow along any edge with weight more than Z, since if it did, then it must carry at least 1 flow along that edge, bringing its total cost to more than Z, contradicting the fact that $\mathsf{opt}_G(b) \le \mathsf{opt}_T(b) = Z$. It follows that removing edges with weight more than Z does not affect the optimal transshipment cost.

Since $|b_v| \le n^C$ for all $v \in V$, it is also well-known that the optimal flow f satisfies $|f_e| \le n^C$ for all $e \in E$. Consider the same flow f on $\widehat{G}$ instead of G; since each edge has its weight increased by Z/n^{C+4}, the total increase in cost of the flow f on $\widehat{G}$ is

$$\sum_{e \in E} |f_e| \cdot \frac{Z}{n^{C+5}} \le \binom{n}{2} \cdot n^C \cdot \frac{Z}{n^{C+5}} \le \frac{Z}{n^3} \overset{(7.30)}{\le} \frac{\mathsf{opt}_G(b)}{n^2}.$$

The cost of the optimal flow on $\widehat{G}$ can only be lower, which proves (7.31). □

7.10.2 Proof of Lemma 7.3.12

Lemma: Restatement of Lemma 7.3.12

Let $G = (V, E)$ be a graph with n vertices and m edges, and let $\mathcal{A}$ be an algorithm that inputs any vertex set $S \subseteq V$ and outputs a $(1 + 1/\log n)$-approximate S-SSSP potential of G. Then, there is an algorithm that computes an ℓ_1-embedding of G into $O(\log^2 n)$ dimensions with distortion $O(\log^3 n)$ and calls $\mathcal{A}$ at most $O(\log^2 n)$ times, plus $\tilde{O}(m)$ additional work and polylog(n) additional time.

Algorithm 15 `L1_embed`$(G = (V, E))$

```
1: Let $N \leftarrow O(\log n)$, $T \leftarrow \lceil \log n \rceil$, $\epsilon \leftarrow 1/\log n$
2: for independent iteration $i = 1, 2, \ldots, N$ do
3:     for $t = 1, 2, \ldots, T$ do
4:         Sample each vertex in $G$ independently with probability $1/2^t$; let $S$ be the sampled set
5:         Compute $(1+\epsilon)$-approximate $S$-SSSP potential $\phi_{i,t}(u)$ of $G$ through algorithm $\mathcal{A}$
6:         Extend $\phi_{i,t}$ so that $\phi_{i,t}(v) = \phi_{i,t}(s)$ for all $v \in S$, so that $\phi_{i,t}(v)$ is now defined for all $v \in V$
7: For each $v \in V$, output the vector $x(v) := \langle \frac{1}{NT}\phi_{i,t}(v) \rangle_{i \in [N], t \in [T]} \in \mathbb{R}^{[N] \times [T]}$ as the $\ell_1$-embedding of $v$
```

Fix two vertices u, v throughout the proof, and define $d := d(u, v)$; we need to show that w.h.p.,

$$\|x(u) - x(v)\|_1 = \frac{1}{NT} \sum_{i \in [N], t \in [T]} |\phi_{i,t}(u) - \phi_{i,t}(v)| \in \left[\frac{d}{O(\log^3 n)},\ d \right].$$

The upper bound is easy: by definition of approximate s-SSSP potential, we have $|\phi_{i,t}(u) - \phi_{i,t}(v)| \le d$ for all i, t, so taking the average over all i, t gives $\frac{1}{NT} \sum_{i,t} |\phi_{i,t}(u) - \phi_{i,t}(v)| \le d$.

To finish Lemma 7.3.12, it remains to prove the lower bound, whose proof occupies the rest of this section.

Lemma 7.10.3

There is a value of $t \in [T]$ such that with probability $\Omega(1)$,

$$|\phi_{i,t}(u) - \phi_{i,t}(v)| \geq \Omega(\epsilon d).$$

For each positive r, define $B(u,r) := \{w \in V : d(w,u) \leq r\}$ as the vertices within distance r from u. Similarly, define $B(v,r)$ as the vertices within distance r from v.

Claim 7.10.4

There exists a value $r \in [d/6, d/3]$ and a (universal) constant $C > 1$ such that

$$|B(u,(1+2\epsilon)r)| \leq C|B(v,r)| \quad \text{or} \quad |B(v,(1+2\epsilon)r)| \leq C|B(u,r)|.$$

Proof. First, we show that such a value r must exist. If not, then we have the chain of inequalities

$$\begin{aligned} |B(v,\frac{d}{6})| &< \frac{1}{C}|B(u,(1+2\epsilon)\frac{d}{6})| < \frac{1}{C^2}|B(v,(1+2\epsilon)^2\frac{d}{6})| < \frac{1}{C^3}|B(u,(1+2\epsilon)^3\frac{d}{6})| \\ &< \cdots < \frac{1}{C^L}|B(u,(1+2\epsilon)^L\frac{d}{6})| \end{aligned}$$

for $L = \lfloor \log_{(1+2\epsilon)} 2 \rfloor = \Theta(1/\epsilon) = \Theta(\log n)$ (we assume w.l.o.g. that the last expression has u and not v). For large enough C, this means that

$$1 \leq |B(u,\frac{d}{6})| \leq \frac{1}{C^{\Theta(\log n)}}|B(u,(1+2\epsilon)^L\frac{d}{6})| < \frac{1}{n}|B(u,(1+2\epsilon)^L\frac{d}{6})| \implies |B(u,(1+2\epsilon)^L\frac{d}{6})| > n$$

which is impossible. Therefore, such a value r exists. $\square$

Take the value r guaranteed by Claim 7.10.4, and assume w.l.o.g. that $|B(u,(1+2\epsilon)r)| \leq C|B(v,r)|$. Pick $t \in [T]$ satisfying $2^{t-1} \leq |B(v,r)| \leq 2^t$, which also means that $|B(u,(1+2\epsilon)r)| \leq O(2^t)$. Suppose we sample each vertex in V with probability $1/2^t$ (line 4). With probability $\Omega(1)$, we sample at least one vertex in $B(v,r)$, and with probability $\Omega(1)$, we sample zero vertices in $B(u,(1+2\epsilon)r)$. Moreover, since $r+(1+2\epsilon)r < 3r \leq 3 \cdot d/3 = d$, the two sets $B(u,(1+2\epsilon)r)$ and $B(v,r)$ are disjoint, so the two events are independent. Thus, with probability $\Omega(1)$, we have both $S \cap B(v,r) \neq \emptyset$ and $S \cap B(u,(1+2\epsilon)r) = \emptyset$, which implies that $d(S,v) \leq r$ and $d(S,u) \geq (1+2\epsilon)r$.

Fix an iteration $i \in [N]$, and let us condition on the previous event. Since $\phi_{i,t}$ is a $(1+\epsilon)$-approximate S-SSSP potential of G, we have $\phi_{i,t}(v) - \phi_{i,t}(s) \leq d(S,v) \leq r$ by property (1) of Definition 7.2.7 and $\phi_{i,t}(u) - \phi_{i,t}(s) \geq \frac{1}{1+\epsilon}d(S,u) \geq \frac{1}{1+\epsilon}(1+2\epsilon)r = (1+\Omega(\epsilon))r$ by Observation 7.2.9. Thus, $|\phi_{i,t}(u) - \phi_{i,t}(v)| \geq \Omega(\epsilon) \cdot r \geq \Omega(\epsilon) \cdot d/6 = \Omega(\epsilon d)$.

Since there are $N = O(\log n)$ trials, w.h.p., one of the iterations $i \in [N]$ will satisfy $|\phi_{i,t}(u) - \phi_{i,t}(v)| \geq \Omega(\epsilon d)$ for the value of t guaranteed by Lemma 7.10.3. Thus, w.h.p., we have

$$\|x(u) - x(v)\|_1 = \frac{1}{NT} \sum_{i,t} |\phi_{i,t}(u) - \phi_{i,t}(v)| \geq \frac{1}{NT} \Omega(\epsilon d) = \frac{d}{O(\log^3 n)},$$

concluding Lemma 7.10.3.

7.10.3 Proof of Lemma 7.5.10

Lemma: Restatement of Lemma 7.5.10

Given a tree $T = (V, E)$ and a set of sources $S \subseteq V$, we can compute an exact S-SSSP potential in $\tilde{O}(m)$ work and polylog(n) time.

Proof. It suffices to compute (exact) S-SSSP distances on T, after which we simply define $\phi(v)$ as the distance to v for each vertex v.

Define a *centroid* of the tree T as a vertex $v \in V$ such that every component of $T - v$ has size at most $|V|/2$. We can compute a centroid r as follows: root the tree T arbitrarily, and for each vertex v, compute the size of the subtree rooted at v; then, let the centroid be a vertex whose subtree has size at least $n/2$, but whose children each have subtrees of size less than $n/2$. Next, compute the distance $d_T(r, S)$ from r to the closest vertex in S, which can be accomplished by computing SSSP on the tree with r as the source. Now root the tree T at r, and for each child vertex v with subtree T_v, construct the following recursive instance: the tree is T_v together with the edge (v, r) of weight $d_T(r, S) + w(v, r)$, and the set S is $(V(T_v) \cap S) \cup \{r\}$. Solve the recursive instances, and for each vertex $u \in V \setminus r$, the distance $d(u)$ is the computed distance in the (unique) recursive instance T_v such that $u \in T_v$.

It is clear that the above algorithm is correct and can be implemented in $\tilde{O}(m)$ work and polylog(n) time. □

7.11 Conclusion

In this chapter, we presented a parallel approximate shortest path algorithm that is optimal up to polylog(n) factors. Unfortunately, the number of polylog(n) factors is prohibitively large (in the dozens), rendering this algorithm far from practical.

Our result was actually achieved concurrently to the one of Andoni, Stein, and Zhong [9], who considered the slightly simpler problem of approximate s–t path. Both results used the same high-level framework and appeared in the same conference (STOC 2020). The algorithm of Andoni et al. also suffers a large number of polylogarithmic factors, although for a different reason.

Lowering the number of logarithmic factors to, say, single digits is an interesting open problem that will require many new insights. Such an algorithm will almost certainly be simpler and more natural, and would likely broaden our understanding of the parallel shortest path problem.

Chapter 8

Deterministic Expander Decomposition

In this chapter, based on [27, 72], we present the deterministic expander decomposition routines that we use in preconditioning-based algorithms for other chapters of the thesis. There are two main variants of expander decompositions that we need, both for weighted, undirected graphs. The first considers additional *custom demands* on the vertices, which are necessary for the deterministic Steiner mincut algorithm of Section 3.3. The second variant does not require vertex demands, but enforces an additional *boundary-linkedness* condition that is necessary for the global mincut application (Chapter 6).

However, for most of the chapter, we will stick with the standard expander decomposition setting on *unweighted* graphs, following the treatment in [27]. At a high level, the algorithm uses the *cut-matching game* technique of Khandekar, Rao, and Vazirani [59], originally developed for the sparsest cut problem. Their original framework is randomized, so instead. we apply a recursive strategy that reduces the cut-matching game setting to smaller instances of itself.

We then use unweighted expander decomposition as a subroutine to solving the weighted setting with custom demands, following the work of [72]. Finally, we augment the weighted expander decomposition algorithm (with standard demands) to handle the additional boundary-linkedness condition.

8.1 Background

In its standard form, an (ϵ, ϕ)-expander decomposition of an unweighted graph $G = (V, E)$ is a partition $\mathcal{P} = \{V_1, \ldots, V_k\}$ of the set V of vertices, such that the graph $G[V_i]$ is a ϕ-expander (see Definition 6.3.6) for all $1 \le i \le k$, and $\sum_{i-1}^{k} \delta_G(V_i) \le \epsilon\mathbf{vol}(G)$. This decomposition was introduced in [43, 53] and has been used as a key tool in many applications, including the ones mentioned in this chapter.

Spielman and Teng [100] provided the first near-linear time algorithm, whose running time is $\tilde{O}(m/\text{poly}(\epsilon))$, for computing a *weak* variant of the $(\epsilon, \epsilon^2/\text{poly}(\log n))$-expander decomposition, where, instead of ensuring that each resulting graph $G[V_i]$ has high conductance, the guarantee is that for each such set V_i there is some larger set W_i of vertices, with $V_i \subseteq W_i$, such that $\Phi(G[W_i]) \geq \epsilon^2/\text{poly}(\log n)$. This caveat was first removed in [87], who showed an algorithm for computing an $(\epsilon, \epsilon/n^{o(1)})$-expander decomposition in time $O(m^{1+o(1)})$ (we note that [104] provided similar results with somewhat weaker parameters). More recently, [94] provided an algorithm for computing $(\epsilon, \epsilon/\text{poly}(\log n))$-expander decomposition in time $\tilde{O}(m/\epsilon)$. Unfortunately, all algorithms mentioned above are randomized.

The only previous subquadratic-time deterministic algorithm for computing an expander decompositions is implicit in [41], running in time $O(m^{1.5+o(1)})$.

Weighted Graphs with Additional Requirements. Recently, a number of results have been discovered that require an expander decomposition routine on weighted graphs with additional requirements. These include the deterministic mincut algorithm of Chapter 6 and the deterministic Steiner mincut algorithm of Section 3.3. Since the unweighted case remains the main contribution of this chapter and does not require definitions or techniques from these weighted variants, we defer their details to Sections 8.7 and 8.8

8.1.1 Our Techniques: Unweighted

Our unweighted deterministic expander decomposition algorithm uses a core subroutine that we name BalCutPrune, which will become the main focus for most of this chapter.

Definition 8.1.1: BalCutPrune problem

The input to the α-approximate BalCutPrune problem is a graph $G = (V, E)$, a conductance parameter $0 < \phi \leq 1$, and an approximation factor α. The goal is to compute a cut (A, B) in G, with $|E_G(A, B)| \leq \alpha\phi \cdot \mathbf{vol}(G)$, such that one of the following holds: either

1. **(Cut)** $\mathbf{vol}_G(A), \mathbf{vol}_G(B) \geq \mathbf{vol}(G)/3$; or
2. **(Prune)** $\mathbf{vol}_G(A) \geq \mathbf{vol}(G)/2$, and graph $G[A]$ has conductance at least ϕ.

BalCutPrune is a simpler problem to study, and the reduction from expander decomposition to BalCutPrune is straightforward: start with the original graph, and iteratively call BalCutPrune on any cluster that is not a certified ϕ-expander to split it into two smaller clusters. We formalize this algorithm and reduction in Section 8.6.

The main technical result of this chapter is a deterministic algorithm for BalCutPrune.

Theorem 8.1.2

There is a deterministic algorithm, that, given a graph G with m edges, and parameters $\phi \in (0, 1]$, $1 \leq r \leq O(\log n)$, and $\alpha = (\log m)^{r^2}$, computes a solution to the α-approximate BalCutPrune problem instance (G, ϕ) in time $O\left(m^{1+O(1/r)+o(1)} \cdot (\log m)^{O(r^2)}/\epsilon^2\right)$.

Our algorithm for the proof of Theorem 8.1.2 is based on the *cut-matching game* framework that was introduced by Khandekar, Rao and Vazirani [59], and has been used in numerous algorithms for computing sparse cuts [41, 59, 87, 94] and beyond (e.g. [21, 22, 26, 91]). Intuitively, the cut-matching game consists of two algorithms: one algorithm, called the *cut player*, needs to compute a balanced cut of a given graph that has a small value, if such a cut exists. The second algorithm, called the *matching player*, needs to solve (possibly approximately) a single-commodity maximum flow / minimum cut problem. A combination of these two algorithms is then used in order to compute a sparse cut in the input graph, or to certify that no such cut exists. Unfortunately, all current algorithms for the cut player are randomized. Our main technical contribution is an efficient deterministic algorithm that implements the cut player. The algorithm itself is recursive, and proceeds by recursively running many cut-matching games in parallel, on much smaller graphs. This requires us to adapt the algorithm of the matching player, so that it solves a somewhat harder multi-commodity flow problem. We now provide more details on the cut-matching game and on our implementation of it.

Overview of the Cut-Matching Game. We start with a high-level overview of a variant of the cut-matching game, due to Khandekar et al. [58]. We say that a graph W is a ψ-*expander* if it has no cut of sparsity less than ψ. We will informally say that W is an *expander* if it is a ψ-expander for some $\psi = 1/n^{o(1)}$. Given a graph $G = (V, E)$, the goal of the cut-matching game is to either find a balanced and sparse cut in G, or to embed an expander $W = (V, E')$ (called a *witness*) into G; note that W and G are defined over the same vertex set. The embedding of W into G needs to map every edge e of W to a path P_e in G connecting the endpoints of e. The *congestion* of this embedding is the maximum number of paths in $\{P_e \mid e \in E(W)\}$ that share a single edge of G. We require that the congestion of the resulting embedding is low. Such an embedding serves as a certificate that there is no sparse balanced cut in G. This follows from the fact that, if W is a ψ-expander, and it has a low-congestion embedding into another graph G, then G itself is a ψ'-expander, where ψ' depends on ψ and on the congestion of the embedding. The algorithm proceeds via an interaction between two algorithms, the cut player, and the matching player, and consists of $O(\log n)$ *rounds*.

At the beginning of every round, we are given a graph W whose vertex set is V, and its embedding into G; at the beginning of the first round, W contains the set V of vertices and no edges. In every round, the cut player either:

(C1) "cuts W", by finding a balanced sparse cut S in W; or

(C2) "certifies W" by announcing that W is an expander.

If W is certified (Item (C2)), then we have constructed the desired embedding of an expander into G, so we can terminate the algorithm and certify that G has no balanced sparse cut. If a cut S is found in W (Item (C1)), then we invoke the matching player, who either:

(M1) "matches W", by adding to W a large matching $M \subseteq S \times (V \setminus S)$ that can be embedded into G with low congestion; or

(M2) "cuts G", by finding a balanced sparse cut T in G (the cut T is intuitively what prevents the matching player from embedding a large matching $M \subseteq S \times (V \setminus S)$ into G).

If a sparse balanced cut T is found in graph G (Item (M2)), then we return this cut and terminate the algorithm. Otherwise, the game continues to the next round. It was shown in [58] that the algorithm must terminate after $\Theta(\log n)$ rounds.

In the original cut-matching game by Khandekar, Rao and Vazirani [59], the matching player was implemented by an algorithm that computes a single-commodity maximum flow / minimum cut. The algorithm for the cut player was defined somewhat differently, in that in the case of Item (C1), the cut that it produced was not necessarily sparse, but it still had some useful properties, which guaranteed that the algorithm terminates after $O(\log^2 n)$ iterations. In order to implement the cut player, the algorithm of [59] (implicitly) considers n vectors of dimension n each, that represent the probability distributions of random walks on the witness graph, starting from different vertices of G, and then uses a random projection of these vectors in order to construct the balanced cut. The algorithm exploits the properties of the witness graph in order to compute these projections efficiently, without explicitly constructing these vectors, which would be too time consuming. Previous work (see, e.g., [25, 94]) implies that one can use algorithms for computing *maximal* flows instead of maximum flows in order to implement the matching player in near-linear time deterministically, if the target parameters $1/\phi, \alpha \leq n^{o(1)}$. This still left open the question: *can the cut player be implemented via a deterministic and efficient algorithm?*

A natural strategy for derandomizing the algorithm of [59] for the cut player is to avoid the random projection of the vectors. In a previous work of a subset of the authors with Yingchareonthawornchai [41], this idea was used to develop a fast PageRank-based algorithm for the cut player, that can be viewed as a derandomization of the algorithm of Andersen, Chung and Lang for balanced sparse cut [8]. Unfortunately, it appears that this technique cannot lead to an algorithm whose running time is below $\Theta(n^2)$: if we cannot use random projections, then we need to deal with n vectors of dimension n each when implementing the cut player, and so the running time of $\Omega(n^2)$ seems inevitable. In this chapter, we implement the cut player in a completely different way from the previously used approaches, by solving the balanced sparse cut problem recursively.

We start by observing that, in order to implement the cut player via the approach of [58], it is sufficient to provide an algorithm for computing a balanced sparse cut on the witness graph W; in fact, it is not hard to see that it is sufficient to solve this problem approximately. However, this leads us to a chicken-and-egg situation, where, in order to solve the $\mathsf{BalCutPrune}$ problem on the input graph G, we need to solve the $\mathsf{BalCutPrune}$ problem on the witness graph W. While graph W is guaranteed to be quite sparse (with maximum vertex degree $O(\log n)$), it is not clear that solving the $\mathsf{BalCutPrune}$ problem on this graph is much easier.

This motivates our recursive approach, in which, in order to solve the $\mathsf{BalCutPrune}$ problem on the witness graph W, we run a large number of cut-matching games in it simultaneously, each of which has a separate witness graph, containing significantly fewer vertices. It is then sufficient to solve the BalCutPrune problem on each of the resulting, much smaller, witness graphs. We prove the following theorem that provides a deterministic algorithm for the cut player via this recursive approach.

Theorem 8.1.3

There is an universal constant N_0, and a deterministic algorithm, that we call CutOrCertify, that, given an n-vertex graph $G = (V, E)$ with maximum vertex degree $O(\log n)$, and a parameter $r \geq 1$, such that $n^{1/r} \geq N_0$, returns one of the following:

- either a cut (A, B) in G with $|A|, |B| \geq n/4$ and $|E_G(A, B)| \leq n/100$; or
- a subset $S \subseteq V$ of at least $n/2$ vertices, such that $\Psi(G[S]) \geq 1/\log^{O(r)} n$.

The running time of the algorithm is $O\left(n^{1+O(1/r)} \cdot (\log n)^{O(r^2)}\right)$.

We note that a somewhat similar recursive approach was used before, e.g., in Madry's construction of j-trees [78], and in the recursive construction of short cycle decompositions [24, 76]. In fact, [41] use Madry's j-trees to solve BalCutPrune by running cut-matching games on graphs containing fewer and fewer nodes, obtaining an $(m^{1.5+o(1)})$-time algorithm. Unfortunately, improving this bound further does not seem viable via this approach, since the total number of edges contained in the graphs that belong to deeper recursive levels is very large. Specifically, assume that we are given an n-node graph G with m edges, together with a parameter $k \geq 1$. We can then use the j-trees in order to reduce the problem of computing BalCutPrune on G to the problem of computing BalCutPrune on k graphs, each of which contains roughly n/k nodes. Unfortunately, each of these graphs may have $\Omega(m)$ edges. Therefore, the total number of edges in all resulting graphs may be as large as $\Omega(mk)$, which is one of the major obstacles to obtaining faster algorithms for BalCutPrune using j-trees.

We now provide a more detailed description of the new recursive strategy that we use in order to prove Theorem 8.1.3.

New Recursive Strategy. We partition the vertices of the input n-vertex graph G into k subsets $V_1, V_2, \ldots, V_k$ of roughly equal cardinality, for a large enough parameter k (for example, $k = n^{o(1)}$). The algorithm consists of two stages. In the first stage, we attempt to construct k expander graphs $W_1, \ldots, W_k$, where $V(W_i) = V_i$ for all $1 \leq i \leq k$, and embed them into the graph G simultaneously. If we fail to do so, then we will compute a sparse balanced cut in G. In order to do so, we run k cut-matching games in parallel. Specifically, we start with every graph W_i containing the set V_i of vertices and no edges, and then perform $O(\log n)$ iterations. In every iteration, we run the CutOrCertify algorithm on each graph $W_1, \ldots, W_k$ in parallel. Assume that for all $1 \leq i \leq k$, the algorithm returns a sparse balanced cut (A_i, B_i) in W_i. We then use an algorithm of the matching player, that either computes, for each $1 \leq i \leq k$, a matching M_i between vertices of A_i and B_i, and computes a low-congestion embedding of all matchings $M_1, \ldots, M_k$ into graph G simultaneously, or it returns a sparse balanced cut in G. In the former case, we augment each graph W_i by adding the set M_i of edges to it. In the latter case, we terminate the algorithm and return the sparse balanced cut in graph G as the algorithm's output. If the algorithm never terminates with a sparse balanced cut, then we are guaranteed that, after $O(\log n)$ iterations, the graphs $W_1, \ldots, W_k$ are all expanders (more precisely, each of these graphs contains a large enough expander, but we ignore this technicality in this informal overview), and moreover, we obtain a low-congestion embedding of the disjoint union of these graphs into G. Note that, in order to execute this stage, we recursively apply algorithm CutOrCertify to k graphs, whose sizes are significantly smaller than the size of the graph G.

In the second stage, we attempt to construct a single expander graph W^* on the set $\{v_1, \ldots, v_k\}$ of vertices, where for each $1 \leq i \leq k$, we view vertex v_i as representing the set V_i of vertices of G. We also attempt to embed the graph W^* into G, where every edge $e = (v_i, v_j)$ is embedded into $\Omega(n/k)$ paths connecting vertices of V_i to vertices of V_j. In order to do so, we start with the graph W^* containing the set $\{v_1, \ldots, v_k\}$ of vertices and no edges and then iterate. In every iteration, we run algorithm CutOrCertify on the current graph W^*, obtaining a partition (A, B) of its vertices. We then use an algorithm of the matching player in order to compute a matching M between vertices of A and vertices of B, and to embed every edge $(v_i, v_j) \in M$ of the matching into $\Omega(n/k)$ paths connecting vertices of V_i to vertices of V_j in graph G, with low congestion. If we do not succeed in computing the matching and the embedding, then the algorithm of the matching player returns a sparse balanced cut in graph G. We then terminate the algorithm and return this cut as the algorithm's output. Otherwise, we add the edges of M to graph W^* and continue to the next iteration. The algorithm terminates once graph W^* is an expander, which must happen after $O(\log n)$ iterations.

Lastly, we compose the expanders $W_1, \ldots, W_k$ and W^* in order to obtain an expander graph $\hat{W}$ that embeds into G with low congestion; the embedding is obtained by combining the embeddings of the graphs $W_1, \ldots, W_k$ and the embedding of graph W^*. This serves as a certificate that G is an expander graph.

Note that the algorithm for the matching player that we need to use differs from the standard one in that it needs to compute k different matchings between k different pre-specified pairs of vertex subsets. Specifically, the algorithm for the matching player is given k pairs $(A_1, B_1), \ldots, (A_k, B_k)$ of subsets of vertices of G of equal cardinality. Ideally, we would like the algorithm to either (i) compute, for all $1 \leq i \leq k$, a perfect matching M_i between vertices of A_i and vertices of B_i, and embed all edges of $M_1 \cup \cdots \cup M_k$ into G simultaneously with low congestion; or (ii) compute a sparse balanced cut in G. In fact our algorithm for the matching player achieves a somewhat weaker objective: namely, the matchings M_i are not necessarily perfect matchings, but they are sufficiently large. In order to overcome this difficulty, we introduce "fake" edges that augment each matching M_i to a perfect matching. As a result, if the algorithm fails to compute a sparse balanced cut in

G, then we are only guaranteed that $G \cup F$ is an expander, where F is (a relatively small) set of fake edges. We then use a known "expander trimming" algorithm of [94] in order to find a large subset $S \subseteq V(G)$ of vertices, such that $G[S]$ is an expander, and the cut S is sufficiently sparse. We note that the notion of fake edges was used before in the context of the cut-matching game, e.g. in [59].

The algorithm of the matching player builds on the idea of Chuzhoy and Khanna [25] of computing maximal sets of short edge-disjoint paths, which can be implemented efficiently via Even-Shiloach's algorithm for decremental single-source shortest paths [36]. Unfortunately, this approach requires slightly slower running time of $O\left(m^{1+O(1/r)} \cdot (\log m)^{O(r^2)}/\phi^2\right)$, introducing a quadratic dependence on $1/\phi$, where ϕ is the conductance parameter. The expander trimming algorithm of [94] that is exploited by the cut player also unfortunately introduces a linear dependence on $1/\phi$. As a result, we obtain an algorithm for the BalCutPrune problem that is sufficiently fast in the high-conductance regime, that is, where $\phi = 1/\text{poly}\log n$, but is too slow for the setting where the parameter ϕ is low. Luckily, the high-conductance regime is sufficient for many of our applications, and in particular it allows us to obtain efficient approximation algorithms for maximum flow. This algorithm can then in turn be used in order to implement the matching player, even in the low-conductance regime, removing the dependence of the algorithm's running time on ϕ. Additional difficulty for the low-conductance regime is that we can no longer afford to use the expander trimming algorithm of [94]. Instead, we provide an efficient deterministic bi-criteria approximation algorithm for the most-balanced sparsest cut problem, and use this algorithm in order to solve the BalCutPrune problem in the low-conductance regime. This part closely follows ideas of [20, 25, 87, 104].

8.1.2 Chapter Organization

We start with additional preliminaries in Section 1.3.

For expander decomposition in the standard, unweighted case, we define the problem to be solved by the new matching player in Section 8.3, and then provide an algorithm for solving it. We also provide a faster algorithm the case where $k = 1$ (that is, the problem of the standard matching player), which we exploit later. We prove our main technical result, Theorem 8.1.3, in Section 8.4, obtaining the algorithm for the cut player. We then prove Theorem 8.1.2 in Section 8.5, and then conclude with Theorem 8.6.1 in Section 8.6.

For the weighted, custom demand setting,

8.2 Additional Preliminaries

8.2.1 Explicit Construction of Expanders

Throughout the algorithm, we will need explicit constructions of constant-degree expanders on any number of vertices. We state and prove the quick lemma below.

Lemma 8.2.1: Explicit expanders

Given integer n, we can construct in linear time an α_0-expander X for some constant $\alpha_0 > 0$, such that every vertex in X has degree at most 9.

Proof. We assume that $n \geq 10$, as otherwise the graph H_n with the required properties can be constructed in constant time. We use the expander construction of Margulis [79] and Gabber and Galil [38]. For an integer $k > 1$, let H'_{k^2} be a graph whose vertex set is set $\mathbb{Z}_k \times \mathbb{Z}_k$ where $\mathbb{Z}_k = \mathbb{Z}/k\mathbb{Z}$. Each vertex $(x, y) \in \mathbb{Z}_k \times \mathbb{Z}_k$ has exactly eight adjacent edges, connecting it to the vertices $(x \pm 2y, y), (x \pm (2y+1), y), (x, y \pm 2x)$, and $(x, y \pm (2x+1))$. Gabber and Galil [38] showed that $\Psi(H'_{k^2}) = \Omega(1)$.

Given a parameter $n \geq 10$, we let k be the unique integer with $(k-1)^2 < n \leq k^2$, and let $n' = n - (k-1)^2$. Clearly, $n' \leq k^2 - (k-1)^2 \leq 2k < (k-1)^2$. In order to obtain the graph H_n, we start with the graph $H_{(k-1)^2}$, whose vertex set we denote by V', and then add a set V'' of n' isolated vertices to this graph. Lastly, we add an arbitrary matching, connecting every vertex of V'' to a distinct vertex of V', obtaining the final graph H_n. It is immediate to verify that $|V(H_n)| = n$, that every vertex in H has degree at most 9, and that H_n is an $\Omega(1)$-expander. $\square$

8.2.2 The Cut-Matching Game

The *cut-matching game* was introduced by Khandekar, Rao, and Vazirani [59] as part of their fast randomized algorithm for the Sparsest Cut and Balanced Cut problems. We use a variation of this game, due to Khandekar et al. [58], that we slightly modify to fit our framework. The game involves two players - the *cut player*, who wants to construct an expander fast, and the *matching player*, who wants to delay the construction of the expander. Initially, the game starts with a graph H that contains an even number n of vertices and no edges. The game is played in iterations, where in every iteration i, some set M_i of edges is added to the current graph H. The ith iteration is played as follows. The cut player computes a partition (A_i, B_i) of $V(H)$ with $|A_i|, |B_i| \geq n/4$ and $|E_H(A_i, B_i)| \leq n/100$. Assume without loss of generality that $|A_i| \leq |B_i|$. The matching player computes any partition (A'_i, B'_i) of $V(H)$ with $|A'_i| = |B'_i|$, such that $A_i \subseteq A'_i$, and then computes an arbitrary perfect matching M_i between A'_i and B'_i. The edges of M_i are then added to the graph H. The algorithm terminates when graph H no longer contains a partition (A, B) of $V(H)$ with $|A|, |B| \geq n/4$ and $|E_H(A, B)| \leq n/100$. Intuitively, once the algorithm terminates, it is easy to see that H contains a large subgraph that is an expander. Alternatively, it is easy to turn H into an expander by adding one last set of $O(n)$ edges to it. We note that the graph H is a multi-graph, that is, it may contain parallel edges. The following theorem follows from the result of [58] (since we slightly modify their setting, we include the proof in Appendix for completeness).

Theorem 8.2.2

There is a constant c_{CMG}, such that the algorithm described above terminates after at most $c_{\text{CMG}} \log n$ iterations.

We will use this cut-matching game together with algorithm CutOrCertify from Theorem 8.1.3, that will be used in order to implement the cut player. The matching player will be implemented by a different algorithm, that we discuss in the following section. Note that, as long as the algorithm from Theorem 8.1.3 produces a cut (A, B) of H with the required properties, we can use the output of this algorithm as the response of the cut player. Theorem 8.2.2 guarantees that, after at most $O(\log n)$ iterations of the game, the algorithm from Theorem 8.1.3 will return a subset $S \subseteq V(H)$ of at least $n/2$ vertices, such that graph $H[S]$ is an expander. Once this happens, we will terminate the cut-matching game.

8.2.3 Expander Pruning

We use the following theorem from [94].

Theorem 8.2.3: Restatement of Theorem 1.3 from [94]

There is a deterministic algorithm, that, given a graph $G = (V, E)$ of conductance $\Phi(G) = \phi$, for some $0 < \phi \leq 1$, and a collection $E' \subseteq E$ of $k \leq \phi|E|/10$ edges of G, computes a subgraph $G' \subseteq G \setminus E'$, that has conductance $\Phi(G') \geq \phi/6$. Moreover, if we denote $A = V(G')$ and $B = V(G) \setminus A$, then $|E_G(A, B)| \leq 4k$, and $\mathbf{vol}_G(B) \leq 8k/\phi$. The total running time of the algorithm is $\tilde{O}(|E|/\phi)$.

We note that [94] provide a significantly stronger result, where the edges of E' arrive in an online fashion and the graph G' is maintained after each edge arrival. Additionally, the running time of the algorithm is $\tilde{O}(k/\phi^2)$ if the algorithm is given an access to the adjacency list of G. However, the weaker statement above is cleaner and it is sufficient for our purposes.

8.2.4 Embeddings of Graphs and Expansion

Next, we define embeddings of graphs, that will be later used to certify graph expansion.

Definition 8.2.4: Graph embedding

Let G, H be two graphs with $V(G) = V(H)$. An *embedding* of H into G is a collection $\mathcal{P} = \{P(e) \mid e \in E(H)\}$ of paths in G, such that for each edge $e \in E(H)$, path $P(e)$ connects the endpoints of e in G. We say that the embedding causes *congestion* η iff every edge $e' \in E(G)$ participates in at most η paths in $\mathcal{P}$.

Next we show that, if G and H are any two graphs with $|V(G)| = |V(H)|$, and H is a ψ-expander that embeds into G with a small congestion, then G is also an expander, for an appropriately chosen expansion parameter. We note that this observation was used in a number of previous algorithms in order to certify that a given graph is an expander; see, e.g. [10, 11, 58, 59, 66, 95].

Lemma 8.2.5

Let G, H be two graphs with $V(G) = V(H)$, such that H is a ψ-expander, for some $0 < \psi < 1$. Assume that there exists an embedding $\mathcal{P} = \{P(e) \mid e \in E(H)\}$ of H into G with congestion at most η, for some $\eta \geq 1$. Then G is a ψ'-expander, for $\psi' = \psi/\eta$.

Proof. Consider any partition (A, B) of $V(G)$, and assume that $|A| \leq |B|$. Consider the corresponding cut (A, B) in H, and let $E' = E_H(A, B)$. Since H is a ψ-expander, $|E'| \geq \psi|A|$. Note that for every edge $e \in E'$, its corresponding path $P(e)$ in G must contain an edge of $E_G(A, B)$. Since the paths in $\mathcal{P}$ cause congestion at most η, we get that $|E_G(A, B)| \geq \frac{|E_H(A,B)|}{\eta} \geq \frac{\psi|A|}{\eta}$. □

8.2.5 Embeddings with Fake Edges and Expansion

In general, when using the cut-matching game, one can usually either embed an expander into a given graph G, or compute a sparse cut S in G. Unfortunately, it is possible that $|S|$ is quite small in the latter case. Since each execution of the cut-matching game algorithm takes at least $\Omega(|E(G)|)$ time, we cannot afford to iteratively remove such small sparse cuts from G, if our goal is to either embed a large expander or to compute a balanced sparse cut in G in almost-linear time. In order to overcome this difficulty, we use *fake edges* (that were also used in [59]), together with the expander pruning algorithm from Theorem 8.2.3.

Specifically, suppose we are given any graph $G = (V, E)$, and let F be a collection of edges whose endpoints lie in V, but the edges of F do not necessarily belong to G. We denote by $G + F$ the graph obtained by adding the edges of F to G. If an edge e lies both in E and F, then we add a new parallel copy of this edge. We note that F is allowed to be a multi-set, in which case multiple parallel copies of an edge may be added to G.

We show that, if H is an expander graph, and we embed it into a graph $G + F$ with a small collection F of fake edges, then we can efficiently compute a large subgraph of G that is an expander.

Lemma 8.2.6

Let G be an n-vertex graph, and let H be another graph with $V(H) = V(G)$, with maximum vertex degree Δ_H, such that H is a ψ-expander, for some $0 < \psi < 1$. Let F be any set of k fake edges for G, and let Δ_G be the maximum vertex degree in $G + F$. Assume that there exists an embedding $\mathcal{P} = \{P(e) \mid e \in E(H)\}$ of H into $G+F$, that causes congestion at most η, for some $\eta \geq 1$. Assume further that $k \leq \frac{\psi n}{32\Delta_G \eta}$. Then there is a subgraph $G' \subseteq G$ that is a ψ'-expander, for $\psi' \geq \frac{\psi}{6\Delta_G \cdot \eta}$, such that, if we denote by $A = V(G')$ and $B = V(G) \setminus A$, then $|A| \geq n - \frac{4k\eta}{\psi}$ and $|E_G(A, B)| \leq 4k$. Moreover, there is a deterministic algorithm, that we call ExtractExpander, that, given $G, H, \mathcal{P}$ and F, computes such a graph G' in time $\tilde{O}(|E(G)|\Delta_G \cdot \eta/\psi)$.

Proof. For convenience, we denote $\hat{G} = G + F$. From Lemma 8.2.5, graph $\hat{G}$ is a $\hat{\psi}$-expander, for $\hat{\psi} = \psi/\eta$. Moreover,

$$\Phi(\hat{G}) \geq \frac{\Psi(\hat{G})}{\Delta_G} \geq \frac{\psi}{\Delta_G \cdot \eta}.$$

In the remainder of the proof, we apply Theorem 8.2.3 to graph $\hat{G}$ and the set F of edges. Recall that the set F of fake edges has cardinality $k \leq \frac{\psi n}{32\Delta_G \cdot \eta} \leq \frac{n \cdot \Phi(\hat{G})}{10} \leq \frac{|E(\hat{G})| \cdot \Phi(\hat{G})}{10}$. Therefore, we can use Theorem 8.2.3 to obtain a subgraph $G' \subseteq (\hat{G} \setminus F) \subseteq G$, that has conductance at least $\frac{\Phi(\hat{G})}{6} \geq \frac{\psi}{6\Delta_G \cdot \eta}$. Denoting $A = V(G')$ and $B = V(\hat{G}) \setminus V(G') = V(G) \setminus V(G')$, Theorem 8.2.3 guarantees that $|E_G(A, B)| \leq |E_{\hat{G}}(A, B)| \leq 4k$. Since $\Psi(G') \geq \Phi(G')$, we have that graph G' is a ψ'-expander for $\psi' = \frac{\psi}{6\Delta_G \eta}$. The running time of the algorithm is $\tilde{O}(|E(\hat{G})|/\Phi(\hat{G})) = \tilde{O}(|E(G)|\Delta_G \eta/\psi)$. It remains to show that $|A|$ is sufficiently large.

Recall that Theorem 8.2.3 guarantees that $|E_{\hat{G}}(A, B)| \leq 4k$, while $\mathbf{vol}_{\hat{G}}(B) \leq \frac{8k}{\Phi(\hat{G})} \leq \frac{8k\Delta_G \eta}{\psi}$. In particular, $|B| \leq \frac{8k\Delta_G \eta}{\psi} \leq \frac{n}{2}$, since $k \leq \frac{\psi n}{32\Delta_G \eta}$. Since graph $\hat{G}$ is a $\hat{\psi}$-expander, and $|E_{\hat{G}}(A, B)| \leq 4k$, we conclude that $|B| \leq \frac{|E_{\hat{G}}(A,B)|}{\hat{\psi}} \leq \frac{4k}{\hat{\psi}} \leq \frac{4k\eta}{\psi}$, and so $|A| \geq n - \frac{4k\eta}{\psi}$. $\square$

8.2.6 j-trees

Finally, for the weighted, custom demand setting, we require the concept of j-trees as defined by Madry [78].

Lemma 8.2.7: [78]

There is a deterministic algorithm that, given an edge-weighted graph $G = (V, E, \boldsymbol{w})$ with $|E| = m$ and capacity ratio $U = \frac{\max_{e \in E} \boldsymbol{w}_e}{\max_{e \in E} \boldsymbol{w}_e}$, together with a parameter $t \geq 1$, computes, in time $\tilde{O}(tm)$, a distribution $\{\lambda_i\}_{i=1}^{t}$ over a collection of t edge-weighted graphs $G_1, \ldots, G_t$, where for each $1 \leq i \leq t$, $G_i = (V, E_i, \boldsymbol{w}_i)$, and the following hold:

- for all $1 \leq i \leq t$, graph G_i is an $(\frac{m \log^{O(1)} m \log U}{t})$-tree, whose core contains at most m edges;
- for all $1 \leq i \leq t$, G embeds into G_i with congestion 1; and
- the graph that's the average of these graphs over the distribution, $\tilde{G} = \sum_i \lambda_i G_i$ can be embedded into G with congestion $O(\log m (\log\log m)^{O(1)})$.

Moreover, the capacity ratio of each G_i is at most $O(mU)$.

In particular, Definition 8.2.4 and lemma 8.2.7 imply that, for any cut $(S, V \setminus S)$, we have that $w(E_{G_i}(S, V \setminus S)) \geq w(E_G(S, V \setminus S))$ for all i, and there exists i where $w(E_{G_i}(S, V \setminus S)) \leq \beta \cdot w(E_G(S, V \setminus S))$. This is the fact that we will use in the weighted, custom demand setting.

8.3 Route or Cut: Algorithm for the Matching Player

The goal of this section is to design an algorithm that will be used by the matching player. We use the following definition for routing pairs of vertex subsets.

Definition 8.3.1: Partial routing

Assume that we are given a graph $G = (V, E)$, and disjoint subsets $A_1, B_1, A_2, B_2, \ldots, A_k, B_k$ of its vertices, that we refer to as *terminals*. Assume further that for each $1 \leq i \leq k$, $|A_i| \leq |B_i|$; we denote $|A_i| = n_i$. A *partial routing* of the sets $A_1, B_1, \ldots, A_k, B_k$ consists of:

- A set $M = \bigcup_{i=1}^{k} M_i \subseteq V \times V$ of pairs of vertices, where for each $1 \leq i \leq k$, M_i is a matching between vertices of A_i and vertices of B_i (we emphasize that the pairs $(u, v) \in M_i$ do not necessarily correspond to edges of G); and
- For every pair $(u, v) \in M$ of vertices, a path $P(u, v)$ connecting u to v in G.

We denote the resulting routing by $\mathcal{P} = \{P(u, v) \mid (u, v) \in M\}$ (note that the matching M is implicitly defined by $\mathcal{P}$). We say that the routing $\mathcal{P}$ causes congestion η, if every edge in G belongs to at most η paths in $\mathcal{P}$. The *value* of the routing is $\sum_{i=1}^{k} |M_i|$.

We are now ready to state the main result of this section, which is an algorithm that will be used by the Matching Player. We note that the theorem is a generalization of a similar result that was proved in [25], for the special case where $k = 1$.

Theorem 8.3.2

There is a deterministic algorithm, that, given an n-vertex graph $G = (V, E)$ with maximum vertex degree Δ, disjoint subsets $A_1, B_1, \ldots, A_k, B_k$ of its vertices, where for all $1 \leq i \leq k$, $|A_i| \leq |B_i|$ and $|A_i| = n_i$, and integers $z \geq 0$, $\ell \geq 32\Delta \log n$, computes one of the following:

- either a partial routing of the sets $A_1, B_1, \ldots, A_k, B_k$, of value at least $\sum_i n_i - z$, that causes congestion at most ℓ^2; or
- a cut (X, Y) in G, with $|X|, |Y| \geq z/2$, and $\Psi_G(X, Y) \leq 72\Delta \log n/\ell$.

The running time of the algorithm is $\tilde{O}(\ell^3 k|E(G)| + \ell^2 kn)$.

(We note that the parameter ℓ in the above theorem bounds the lengths of the paths in $\mathcal{P}$, that is, we will ensure that every path in $\mathcal{P}$ contains at most ℓ edges; however, since our algorithm does not rely on this fact, this is immaterial).

Proof. The proof of the theorem immediately follows from the following lemma.

Lemma 8.3.3

There is a deterministic algorithm, that, given an n-vertex graph $G = (V, E)$ with maximum vertex degree Δ, disjoint subsets $A'_1, B'_1, \ldots, A'_k, B'_k$ of its vertices, where for all $1 \leq i \leq k$, $|A'_i| \leq |B'_i|$, and $|A'_i| = n'_i$, and an integer $\ell \geq 32\Delta \log n$, computes one of the following:

- either a partial routing of the sets $A'_1, B'_1, \ldots, A'_k, B'_k$ in G, of value at least $\left(\sum_{i=1}^k n'_i\right) \cdot \frac{8 \log n}{\ell^2}$ and congestion 1; or
- a cut (X, Y) in G, with $|X|, |Y| \geq \left(\sum_{i=1}^k n'_i\right)/2$, and $\Psi_G(X, Y) \leq 72\Delta \log n/\ell$.

The running time of the algorithm is $\tilde{O}(k\ell|E(G)| + kn)$.

Before we prove the lemma, we complete the proof of Theorem 8.3.2 using it. Throughout the algorithm, we maintain the matchings $M_1, \ldots, M_k$, where M_i is a matching between vertices of A_i and vertices of B_i, and a routing $\mathcal{P} = \{P(u, v) \mid (u, v) \in \bigcup_i M_i\}$. Initially, we set $M_i = \emptyset$ for all i, and $\mathcal{P} = \emptyset$. We then iterate. In every iteration, for each $1 \leq i \leq k$, we let $A'_i \subseteq A_i$ and $B'_i \subseteq B_i$ be the subsets of vertices that do not participate in the matching M_i, and we denote $n'_i = |A'_i|$; since $|A_i| \leq |B_i|$, we are guaranteed that $|A'_i| \leq |B'_i|$. We also denote $N' = \sum_i n'_i$. If $N' \leq z$, then we terminate the algorithm, and return the current matchings $M_1, \ldots, M_k$, together with their routing $\mathcal{P}$. Otherwise, we apply Lemma 8.3.3 to graph G and vertex sets $A'_1, B'_1, \ldots, A'_k, B'_k$. If the outcome is a cut (X, Y) in G, with $|X|, |Y| \geq N'/2$, and $\Psi_G(X, Y) \leq 72\Delta \log n/\ell$, then we terminate the algorithm, and return the cut (X, Y). Notice that, since $N' > z$ holds, we are guaranteed that $|X|, |Y| \geq z/2$, as required. Therefore, we assume from now on that, whenever Lemma 8.3.3 is called, it returns a partial routing $((M'_1, \ldots, M'_k), \mathcal{P}')$ of the vertex sets $A'_1, B'_1, \ldots, A'_k, B'_k$, of value at least $\frac{8N' \log n}{\ell^2}$, that causes congestion 1. We then add the paths in $\mathcal{P}'$ to $\mathcal{P}$, and for each

$1 \leq i \leq k$, we add the matching M'_i to M_i, and continue to the next iteration.

The key in the analysis of the algorithm is to bound the number of iterations. For all $j \geq 1$, let N'_j denote the parameter N' at the beginning of iteration j. Then, since Lemma 8.3.3 returns a routing of value at least $\frac{8N'_j \log n}{\ell^2}$, we get that $N'_{j+1} \leq N_j(1-8\log n/\ell^2)$. Therefore, after ℓ^2 iterations, parameter N'_j is guaranteed to fall below z, and the algorithm will terminate. Notice that the congestion of the final routing $\mathcal{P}$ is bounded by the number of iterations, ℓ^2. Moreover, since the running time of each iteration is $\tilde{O}(k\ell|E(G)|+kn)$, the total running time of the algorithm is $\tilde{O}(k\ell^3|E(G)|+kn\ell^2)$. In order to complete the proof of Theorem 8.3.2, it is now enough to prove Lemma 8.3.3.

Proof of Lemma 8.3.3. Our algorithm is very similar to that employed in [25], and consists of two phases. In the first phase, we employ a simple greedy algorithm that attempts to compute a partial routing of sets $A'_1, B'_1, \ldots, A'_k, B'_k$. If the resulting routing contains enough paths then we terminate the algorithm and return this routing. Otherwise, we proceed to the second phase, where we compute the desired cut.

Phase 1: Route. We use a simple greedy algorithm. Initially, we set, for all $1 \leq i \leq k$, $M_i = \emptyset$, and we set $\mathcal{P} = \emptyset$. The algorithm then iterates, as long as there is a path P in G of length at most ℓ, that, for some $1 \leq i \leq k$, connects some vertex $v \in A'_i$ to some vertex $u \in B'_i$. The algorithm computes any such path P, adds (u,v) to M_i, and adds the path P to $\mathcal{P}$, denoting $P = P(u,v)$. We then delete every edge of P from G, and we delete u from A'_i and v from B'_i, and then continue to the next iteration. The algorithm terminates when, for each $1 \leq i \leq k$, every path in the remaining graph G connecting a vertex of A'_i to a vertex of B'_i has length greater than ℓ (or $A'_i = \emptyset$). It is easy to verify that, for each $1 \leq i \leq k$, the final set M_i is a matching between vertices of A'_i and vertices of B'_i, and that $\mathcal{P}$ is a collection of edge-disjoint paths, of length at most ℓ each, containing, for every pair $(u,v) \in \bigcup_i M_i$, a path $P(u,v)$ connecting u to v in G. If $\sum_i |M_i| \geq \left(\sum_{i=1}^k n'_i\right) \frac{8\log n}{\ell^2}$, then we terminate the algorithm, obtaining the desired partial routing. Otherwise, we continue to the second phase, where a cut (X,Y) will be computed.

We implement the algorithm for the first phase by using Even-Shiloach trees.

Lemma 8.3.4: [32, 36]

There is a deterministic data structure, called ES-tree, that, given an unweighted undirected n-vertex graph G undergoing edge deletions, a root node s, and a depth parameter ℓ, maintains, for every vertex $v \in V(G)$ a value $\delta(s,v)$ such that $\delta(s,v) = \mathrm{dist}_G(s,v)$ if $\mathrm{dist}_G(s,v) \leq \ell$ and $\delta(s,v) = \infty$ otherwise (here, $\mathrm{dist}_G(s,v)$ is the distance between s and v in the current graph G). The data structure supports shortest-paths queries: given a vertex v, return a shortest path connecting s to v in G, if $\mathrm{dist}_G(s,v) \leq \ell$, and return ∞ otherwise. The total update time of the data structure is $\tilde{O}(|E(G)|\ell + n)$, and time needed to process each query is $O(|P|)$, where P is the path returned in response to the query.

We construct k graphs $G_1,\ldots,G_k$, where graph G_i is obtained from a copy of G, by adding a source vertex s_i that connects to every vertex in A'_i with an edge, and a destination vertex t_i, that connects to every vertex in B'_i with an edge. For each $1 \le i \le k$, we then maintain an ES-tree in graph G_i, from source s_i, up to depth $\ell+2$. Note that the total update time needed in order to maintain all these ES-trees under edge deletions is $\tilde{O}(\ell k|E(G)|+kn)$. Our algorithm processes the graphs G_i one-by-one. When graph G_i is processed, we perform a number of iterations, as long as $\mathrm{dist}_{G_i}(s_i,t_i) \le \ell+2$. In each such iteration, we perform a shortest-path query in the corresponding ES-tree for vertex t_i, obtaining a path P, of length at most $\ell+2$, connecting s_i to t_i. By discarding the first and the last edge on this path, we obtain a path P' of length at most ℓ, connecting some vertex $v \in A'_i$ to some vertex $u \in B'_i$. We delete all edges on path P' from all copies $G_1,\ldots,G_k$ of the graph G, and we delete v and u from G_i, updating all corresponding ES-trees. Note that the total time to respond to all queries is $O(|E(G)|)$, as whenever a path P is returned, all its edges are deleted from all graphs G_i. Therefore, the total running time of the algorithm is $\tilde{O}(k\ell|E(G)|+kn)$.

Phase 2: Cut. We use the following standard algorithm that follows the ball-growing paradigm.

Claim 8.3.5

There is a deterministic algorithm, that, given an unweighted n'-vertex graph H' with maximum vertex degree at most Δ, and two sets S,T of its vertices, such that every path connecting a vertex of S to a vertex of T in H' has length greater than ℓ, for some parameter $\ell > 1$ computes, in time $O(|E(H')|)$, a cut Z in H', such that:

- $|Z| \le n'/2$;
- either $S \subseteq Z$ or $T \subseteq Z$ hold; and
- $|E_{H'}(Z, V(H') \setminus Z)| < \frac{8\Delta\log n'}{\ell} \cdot |Z|$.

Proof. Let $S_0 = S$, and for all $j > 0$, let S_j contain all vertices of S_{j-1}, and all neighbors of vertices of S_{j-1} in graph H'. We also define $T_0 = T$, and for all $j > 0$, we let T_j contain all vertices of T_{j-1}, and all neighbors of vertices of T_{j-1} in graph H'. We need the following standard observation:

Observation 8.3.6

There is an index $0 \le j < \lceil \ell/4 \rceil$, such that either (i) $|S_{j+1}| < n'/2$ and $|E_{H'}(S_j, V(H') \setminus S_j)| < \frac{8\Delta\log n'}{\ell} \cdot |S_j|$; or (ii) $|T_{j+1}| < n'/2$ and $|E_{H'}(T_j, V(H') \setminus S_j)| < \frac{8\Delta\log n'}{\ell} \cdot |T_j|$.

Proof. Assume for contradiction that the claim is false. Let j' be the smallest index, such that $|S_{j'}| > n'/2$ or $|T_{j'}| > n'/2$. Assume w.l.o.g. that $|T_{j'}| > n'/2$.

Assume first that $j' < \ell/2$. Then for all $1 \le j \le \lceil \ell/4 \rceil$, $|S_j| < n'/2$ must hold (as otherwise, there is a path connecting a vertex of S to a vertex of T, of length at most ℓ). However, from our assumption, for all $0 \le j < \lceil \ell/4 \rceil$, $|E_{H'}(S_j, V(H') \setminus S_j)| > \frac{8\Delta\log n'}{\ell} \cdot |S_j|$.

Since the maximum vertex degree in H' is bounded by Δ, we get that $|S_{j+1} \setminus S_j| \geq \frac{8 \log n'}{\ell} \cdot |S_j|$, and so $|S_{j+1}| \geq |S_j| \left(1 + \frac{8 \log n'}{\ell}\right)$. Overall, we get that $|S_{\lceil \ell/4 \rceil}| \geq |S_0| \cdot \left(1 + \frac{8 \log n'}{\ell}\right)^{\lceil \ell/4 \rceil} > \frac{n'}{2}$, a contradiction.

Assume now that $j' \geq \ell/2$. Then we get that for all $1 \leq j \leq \lceil \ell/4 \rceil$, $|T_j| < n'/2$ must hold. Applying the same reasoning as above to sets T_j, we conclude that $|T_{\lceil \ell/4 \rceil}| \geq n'/2$, a contradiction. $\square$

The algorithm performs two BFS searches in H' simultaneously, one starting from S and another starting from T, until an index j with the properties guaranteed by Observation 8.3.6 is found. If $|S_{j+1}| < n'/2$ and $|E_{H'}(S_j, V(H') \setminus S_j)| < \frac{8\Delta \log n'}{\ell} \cdot |S_j|$, then we return $Z = S_j$; otherwise, and otherwise we return $Z = T_j$. $\square$

We are now ready to describe the algorithm for Phase 2. For convenience, we denote $N = \sum_{i=1}^{k} n'_i$. Recall that Phase 2 is only executed if the routing $\mathcal{P}$ computed in Phase 1 contains fewer than $\frac{8N \log n}{\ell^2}$ paths. Let E' be the set of all edges lying on the paths in $\mathcal{P}$, so $|E'| \leq \frac{8N \log n}{\ell}$ (as the length of every path in $\mathcal{P}$ is at most ℓ), and let $H = G \setminus E'$. We also denote, for all $1 \leq i \leq k$, by $A''_i \subseteq A'_i$ the subset of all vertices of the original set A'_i that do not participate in the matching M_i, and we define $B''_i \subseteq B'_i$ similarly. Notice that for all $1 \leq i \leq k$, if $A''_i, B''_i \neq \emptyset$, then the length of the shortest path, connecting a vertex of A''_i to a vertex of B''_i is greater than ℓ.

Our algorithm is iterative. We maintain a subgraph H' of H, that is initially set to be H. In every iteration i, we compute a subset $U_i \subseteq V(H')$ of vertices of H', such that $|U_i| \leq |V(H')|/2$, and $|E_{H'}(U_i, V(H') \setminus U_i)| < \frac{8\Delta \log n}{\ell} \cdot |U_i|$. We then delete, from graph H', all vertices of U_i, and continue to the next iteration. Throughout the algorithm, we may update the sets A''_j and B''_j, by removing some vertices from them.

The algorithm is executed as long as there is some index $1 \leq j \leq k$, with $A''_j, B''_j \neq \emptyset$, and as long as $|\bigcup_i U_i| \leq n/4$; if either of these conditions do not hold, the algorithm is terminated. We now describe the ith iteration of the algorithm, and we let $1 \leq j \leq k$ be an index for which $A''_j, B''_j \neq \emptyset$. We apply the algorithm from Claim 8.3.5 to the current graph H', and the sets $S = A''_j$, $T = B''_j$ of vertices; recall that every path connecting a vertex of A''_j to a vertex of B''_j in H' has length greater than ℓ. Let Z be the cut returned by the algorithm. We set $U_i = Z$. We also denote by $E_i = E_{H'}(Z, V(H') \setminus Z)$. Recall that we are guaranteed that $|E_i| \leq \frac{8\Delta \log n}{\ell} \cdot |U_i|$. Moreover, either $A''_j \subseteq U_i$, or $B''_j \subseteq U_i$. We update the current graph H', by deleting the vertices of U_i from it. For all $1 \leq j' \leq k$, we delete from $A''_{j'}$ and from $B''_{j'}$ all vertices that lie in the set U_i.

Let q be the number of iterations in the algorithm; it is easy to see that $q \leq k$. Therefore, the running time of the algorithm in Phase 2 so far is $O(k \cdot |E(H)|) = O(k \cdot |E(G)|)$. Let $U = \bigcup_{i=1}^{r} U_i$, and let $\hat{E} = \bigcup_{i=1}^{r} E_i$.

If the algorithm terminated because $|U| \geq n/4$, then we are guaranteed that $|U| \geq N/2$, as $N \leq n/2$ must hold. Otherwise, we are guaranteed that for all $1 \leq j \leq k$, either $A''_j = \emptyset$ (and so $A'_j \subseteq U$), or $B''_j = \emptyset$ (and so $B'_j \subseteq U$). In the latter case, we get that:

$$|U| \geq \sum_{j=1}^{k} n'_j - |\mathcal{P}| \geq N - \frac{8N\log n}{\ell^2} \geq N/2,$$

since we have assumed that $\ell \geq 32\Delta\log n$. Moreover, it is immediate to verify that $|\hat{E}| \leq \frac{8\Delta\log n}{\ell} \cdot |U|$.

Consider now the original graph H. We define a cut (X, Y) in H by setting $X = U$ and $Y = V(H) \setminus U$. Since $|E(G) \setminus E(H)| = |E'| \leq \frac{8N\log n}{\ell} \leq \frac{16|U|\log n}{\ell}$, we get that $|E_G(X, Y)| \leq |\hat{E}| + |E'| \leq \frac{24\Delta\log n}{\ell} \cdot |X|$.

Next, we claim that $|X| \leq 3n/4$. Indeed, we are guaranteed that $\sum_{i=1}^{q-1}|U_i| \leq n/4$, and so $U_q \leq \frac{n - \sum_{i=1}^{q-1}|U_i|}{2}$. We then get that altogether, $|X| = \sum_{i=1}^{q}|U_i| \leq \frac{n}{2} + \frac{\sum_{i=1}^{q-1}|U_i|}{2} \leq \frac{3n}{4}$. In particular, $|Y| \geq n/4$ and so $|Y| \geq |X|/3$. Therefore, $|E_G(X, Y)| \leq \frac{24\Delta\log n}{\ell} \cdot |X| \leq \frac{72\Delta\log n}{\ell} \cdot \min\{|X|, |Y|\}$, and so $\Psi_G(X, Y) \leq \frac{72\Delta\log n}{\ell}$. As observed already, $|X| \geq N/2 = \sum_i n'_i/2$, and $|Y| \geq n/4 \geq \sum_i n'_i/2$, as $\sum_i n'_i \leq n/2$ must hold. □

The following corollary follows immediately from Theorem 8.3.2, by setting the parameter $\ell = 144\Delta\log n/\psi$.

Corollary 8.3.7

There is a deterministic algorithm, that we call RouteOrCut, that, given an n-vertex graph $G = (V, E)$ with maximum vertex degree Δ, disjoint subsets $A_1, B_1, \ldots, A_k, B_k$ of its vertices, where for all $1 \leq i \leq k$, $|A_i| \leq |B_i|$ and $|A_i| = n_i$, an integer $z \geq 0$, and a parameter $0 < \psi < 1/2$, computes one of the following:

- either a partial routing of the sets $A_1, B_1, \ldots, A_k, B_k$, of value at least $\sum_i n_i - z$, that causes congestion at most $O(\Delta^2\log^2 n/\psi^2)$; or
- a cut (X, Y) in G, with $|X|, |Y| \geq z/2$, and $\Psi_G(X, Y) \leq \psi$.

The running time of the algorithm is $\tilde{O}(\Delta^3 k|E(G)|/\psi^3 + k\Delta^2 n/\psi^2)$.

An Improved Algorithm for $k = 1$

For the special case where $k = 1$, we provide a somewhat faster algorithm, summarized in the following theorem. We note that this algorithm is not essential for the proof of our main result (Theorem 8.1.2), but we can use it to provide a self-contained proof of the theorem with a somewhat slower running time, which we believe is of independent interest.

Theorem 8.3.8

There is a deterministic algorithm, that we call RouteOrCut-1Pair, that, given a connected n-vertex m-edge graph $G = (V, E)$ with maximum vertex degree Δ, two disjoint subsets A_1, B_1 of its vertices, where $|A_1| \leq |B_1|$ and $|A_1| = n_1$, an integer $z \geq 0$, and a parameter $0 < \psi < 1/2$, computes one of the following:

- either a partial routing of the sets A_1, B_1, of value at least $n_1 - z$, that causes congestion at most $4\Delta/\psi$; or
- a cut (X, Y) in G, with $|X|, |Y| \geq z/\Delta$, and $\Psi_G(X, Y) \leq \psi$.

The running time of the algorithm is $O\left(\frac{m\Delta \log m}{\psi}\right)$.

Proof. Theorem 8.3.8 is an easy application of either the *bounded-height* variant of the push-relabel-based algorithm of Henzinger, Rao and Wang [51] for max-flow, or the *bounded-height* variant of the blocking-flow-based algorithms by Orrecchia and Zhu [89].[1]

We start by introducing some basic notation. Suppose we are given an unweighted undirected graph $G = (V, E)$. We let $S : V \to \mathbb{Z}_{\geq 0}$ denote a *source function* and $T : V \to \mathbb{Z}_{\geq 0}$ denote a *sink function*. For a vertex $v \in V$, we sometimes call $T(v)$ its *sink capacity*. Intuitively, initially, for every vertex $v \in V$, we have $S(v)$ units of mass (substance that needs to be routed) placed on vertex v. Additionally, every vertex $v \in V$ may absorb up to $T(v)$ units of mass. Our goal is to route the initial mass across the graph (using standard single-commodity flow) so that all mass is absorbed. We use a flow function $f : V \times V \to \mathbb{R}$, that must satisfy: (i) for all $u, v \in V$, $f(u, v) = -f(v, u)$; and (ii) if $(u, v) \notin E$, then $f(u, v) = 0$. Whenever $f(u, v) > 0$, we interpret it as $f(u, v)$ units of mass are sent via the edge (u, v) from u to v, while $f(u, v) < 0$ means that the same amount of mass is sent in the opposite direction.

We require that $\sum_{v \in V} S(v) \leq \sum_u T(u)$, that is, the total amount of mass that needs to be routed is bounded by the total sink capacities of the vertices. Given a flow $f : V \times V \to \mathbb{R}$, the *congestion* of f is $\max_{e \in E} |f(e)|$. We say that f is a *preflow* if, for every vertex $v \in V$, $\sum_{u \in V} f(v, u) \leq S(v)$; in other words, the net amount of mass routed away from any node v is bounded by the amount of the source mass $S(v)$. For every vertex $v \in V$, we also denote by $f(v) = S(v) + \sum_{u \in V} f(u, v)$ the amount of mass that remains at v after the routing f. We define the *absorbed mass* of a node v as $\mathrm{ab}_f(v) = \min\{f(v), T(v)\}$, and the *excess of* v as $\mathrm{ex}_f(v) = f(v) - \mathrm{ab}_f(v)$, measuring the amount of flow that remains at v and cannot be absorbed by it. Note that, if $\mathrm{ex}_f(v) = 0$ for every vertex v, then all the mass is successfully routed to the sinks. Let $\mathrm{ex}_f(V) = \sum_v \mathrm{ex}_f(v)$ denote the total amount of mass that is not absorbed by the sinks.

The following lemma easily follows from Theorem 3.3 in [88] (or Theorem 3.1 in [51]).

[1]Both algorithms are designed to have *local* running time, that is, they may not read the whole graph. However, we do not need to use this property here.

Lemma 8.3.9

There is a deterministic algorithm, that, given an m-edge graph $G = (V, E)$, a source function $S : V \to \mathbb{Z}_{\geq 0}$, a sink function $T : V \to \mathbb{Z}_{\geq 0}$, and a parameter $0 < \phi \leq 1$, such that $\sum_{v \in V} S(v) \leq \sum_{v \in V} T(v)$, and for every vertex $v \in V$, $S(v) \leq \deg_G(v)$ and $T(v) \leq \deg_G(v)$, computes, in time $O\left(\frac{m \log m}{\phi}\right)$, an integral preflow f of congestion at most $4/\phi$. Moreover, if the total excess $\text{ex}_f(V) > 0$, then the algorithm also computes a cut $(S, \overline{S})$ with $\Phi_G(S) < \phi$ and $\mathbf{vol}_G(S), \mathbf{vol}_G(\overline{S}) \geq \text{ex}_f(V)$.

We are now ready to complete the proof of Theorem 8.3.8. For convenience, we denote A_1 by A, B_1 by B, and n_1 by N. For the input graph $G = (V, E)$, we define a source function as follows: for all $v \in A$, $S(v) = 1$, and for all other vertices, $S(v) = 0$. Similarly, we define the sink function to be $T(v) = 1$ if $v \in B$, and $T(v) = 0$ otherwise.

We then apply the algorithm from Lemma 8.3.9 to graph G, source function S, sink function T and parameter $\phi = \psi/\Delta$. Let f be the resulting preflow with congestion at most $4/\phi \leq 4\Delta/\psi$. The running time of the algorithm is $O\left(\frac{m \log m}{\phi}\right) = O\left(\frac{m\Delta \log m}{\psi}\right)$

We now consider two cases. The first case happens when $\text{ex}_f(V) \geq z$. In this case, we obtain a cut (X, Y) with $\Phi_G(X, Y) < \phi$ and $\mathbf{vol}_G(X), \mathbf{vol}_G(Y) \geq \text{ex}_f(V) \geq z$. Since the maximum vertex degree in G is bounded by Δ, we get that $|X|, |Y| \geq z/\Delta$. Moreover, $\Psi_G(X, Y) \leq \Delta \Phi_G(X, Y) \leq \Delta\phi \leq \psi$.

Consider now the second case, where $\text{ex}_f(V) < z$. Let B' be a multi-set of vertices, where for each vertex $v \in V$, we add $\text{ex}_f(v)$ copies of v into B' (since f is integral, so is $\text{ex}_f(v)$ for all $v \in V$). Then $|B'| \leq z$, and f defines a valid integral flow from A to $B \cup B'$, with congestion at most $4\Delta/\psi$, such that all but at most z flow units terminate at distinct vertices of B. It now remains to compute a decomposition of f into flow-paths, and then discard the flow-paths that terminate at vertices of B'. This can be done by using, for example, the link-cut tree [98], or simply a standard Depth-First Search. For the latter, construct a graph G', obtained from G by creating $|f(e)|$ parallel copies of every edge $e \in E(G)$, that are directed along the direction of the flow f on e; recall that $|f(e)| \leq 4\Delta/\psi$. We also add a source s that connects to every vertex of A with a directed edge. We then perform a DFS search of the resulting graph G', starting from s. If the DFS search leaves some vertex v without reaching any vertex of $B \cup B'$, then we delete v from the graph G'. If the search reaches a vertex $v \in B \cup B'$, then we retrace the current path from s to v, adding it to the path-decomposition that we are constructing, and deleting all edges on this path from G'. We then restart the DFS search. It is easy to verify that every edge is traversed at most twice throughout this procedure, and so the total running time is $O|E(G')| = O(|E(G)| \cdot \Delta/\psi)$. Let $\mathcal{P}$ be the final collection of paths that we obtain. Then every vertex of A has exactly one path in $\mathcal{P}$ originating from it, and all but at most z paths in $\mathcal{P}$ terminate at distinct vertices of B. We discard from $\mathcal{P}$ all paths that do not terminate at vertices of B, obtaining the desired final collection of paths. The total running time of the algorithm is $O\left(\frac{m\Delta \log m}{\psi}\right)$. $\square$

8.4 Deterministic Cut-Matching Game: Proof of Theorem 8.1.3

The goal of this section is to prove Theorem 8.1.3. We do so using the following theorem, that can be thought of as a restatement of Theorem 8.1.3 in a way that will be more convenient to work with in our inductive proof. Recall that c_{CMG} is the constant from Theorem 8.2.2.

Theorem 8.4.1

There are universal constants c_0, N_0 and a deterministic algorithm, that, given an n-vertex graph $G = (V, E)$ and parameters N, q with $N > N_0$ an integral power of 2, and $q \geq 1$ an integer, such that $n \leq N^q$, and the maximum vertex degree in G is at most $c_{\text{CMG}} \log n$, computes one of the following:

- either a cut (A, B) in G with $|A|, |B| \geq n/4$ and $|E_G(A, B)| \leq n/100$; or
- a subset $S \subseteq V$ of at least $n/2$ vertices, such that $\Psi(G[S]) \geq 1/\left(q \log N\right)^{8q}$.

The running time of the algorithm is $O\left(N^{q+1} \cdot (q \log N)^{c_0 q^2}\right)$.

We first show that Theorem 8.1.3 follows from Theorem 8.4.1. The parameter N_0 in Theorem 8.1.3 remains the same as that in Theorem 8.4.1. Assume that we are given an n-vertex graph and a parameter r, such that $n^{1/r} \geq N_0$. We set $q = r$, and we let N be the smallest integral power of 2 such that $N \geq n^{1/q}$; observe that $(N/2)^q \leq n \leq N^q$ and $N \geq N_0$ hold. Moreover, since $q \log(N/2) \leq \log n$, if N_0 is a large enough constant, then $q \log N \leq 2 \log n$.

We apply the algorithm from Theorem 8.4.1 to graph G with the parameter q. If the outcome is a cut (A, B) with $|A|, |B| \geq n/4$ and $|E(A, B)| \leq n/100$, then we return this cut as the outcome of the algorithm. Otherwise, we obtain a subset $S \subseteq V$ of at least $n/2$ vertices, such that $\Psi(G[S]) \geq 1/\left(q \log N\right)^{8q} \geq 1/\left(2 \log n\right)^{8q} \geq \Omega\left(1/(\log n)^{O(r)}\right)$, as required. Lastly, the running time of the algorithm is $O\left(N^{q+1} \cdot (q \log N)^{c_0 q^2}\right) = O\left(n^{1+O(1/r)} \cdot (\log n)^{O(r^2)}\right)$.

The remainder of this section is dedicated to proving Theorem 8.4.1. The proof is by induction on the parameter q. We start with the base case where $q = 1$ and then show the step for $q > 1$.

8.4.1 Base Case: $q = 1$

The algorithm uses the following key theorem.

Theorem 8.4.2

There is a deterministic algorithm that, given as input a graph $G' = (V', E')$ with $|V'| = n'$ and maximum vertex degree $\Delta = O(\log n')$, in time $\tilde{O}((n')^2)$ returns one of the following:

- either a subset $S \subseteq V'$ of at least $2n'/3$ vertices such that $G'[S]$ is an $\Omega(1/\log^5 n')$-expander; or
- a cut (X, Y) in G' with $|X|, |Y| \geq \Omega(n'/\log^5 n')$ and $\Psi_{G'}(X, Y) \leq 1/100$.

We prove Theorem 8.4.2 below, after we complete the proof of Theorem 8.1.3 for the case where $q = 1$ using it. Our algorithm performs a number of iterations. We maintain a subgraph $G' \subseteq G$; at the beginning of the algorithm, $G' = G$. In the ith iteration, we compute a subset $S_i \subseteq V(G')$ of vertices, and then update the graph G' by deleting the vertices of S_i from it. The iterations are performed as long as $|\bigcup_i S_i| < n/4$.

In order to execute the ith iteration, we consider the current graph G', denoting $|V(G')| = n'$. Note that, since we assume that $|\bigcup_{i'<i} S_{i'}| < n/4$, we get that $n' \geq 3n/4$. We then apply Theorem 8.4.2 to graph G'. If the outcome is a subset $S \subseteq V'$ of at least $2n'/3$ vertices such that $G'[S]$ is an $\Omega(1/\log^5 n')$-expander, then we terminate the algorithm and return S; in this case we say that the iteration terminated with an expander. Notice that, since $n' \geq 3n/4$, and $|S| \geq 2n'/3$, we are guaranteed that $|S| \geq n/2$. Moreover, assuming that N_0 is a large enough constant, the expansion of $G[S]$ is at least $\Omega(1/\log^5 n') \geq 1/\log^8 n \geq 1/\log^8 N$ as required. Otherwise, we obtain a cut (X, Y) in G' with $|X|, |Y| \geq \Omega(n'/\log^5 n')$ and $\Psi_{G'}(X, Y) \leq 1/100$; in this case we say that the iteration terminated with a cut. Assume w.l.o.g. that $|X| \leq |Y|$. We then set $S_i = X$, update the graph G' by removing the vertices of S_i from it, and continue to the next iteration. If the algorithm does not terminate with an $1/\log^8 N$-expander, then it terminates once $|\bigcup_{i'} S_{i'}| \geq n/4$ holds. Let i denote the number of iterations in this case. Since we are guaranteed that $|\bigcup_{i'<i} S_{i'}| < n/4$, while $|S_i| \leq n'/2 \leq 3n/8$, we get that $n/4 \leq |\bigcup_{i'=1}^{i} S_{i'}| \leq 5n/8$. Let $A = \bigcup_{i'=1}^{i} S_{i'}$, and let $B = V(G) \setminus A$. From the above discussion, we are guaranteed that $|A|, |B| \geq n/4$, and moreover, since the cut $S_{i'}$ that we obtain in every iteration i' has sparsity at most $1/100$ in its current graph G', it is easy to verify that $|E_G(A, B)| \leq |A|/100 \leq n/100$. We then return the cut (A, B) as the algorithm's outcome. Since for all $1 \leq i' \leq i$, $|S_{i'}| \geq \Omega(n/\log^5 n)$, the number of iterations is bounded by $O(\log^6 n)$, and so the total running time of the algorithm is $\tilde{O}(n^2) = O(N^2 \log^{c_0} n)$, if c_0 is a large enough constant. In the remainder of this subsection we focus on the proof of Theorem 8.4.2.

Proof of Theorem 8.4.2

As our first step, we use Lemma 8.2.1 to construct in linear time a ψ^*-expander $H = H_{n'}$ on n' vertices, with $\psi^* = \Psi(H) = \Omega(1)$, such that maximum vertex degree in H is at most 9. This can be done using standard explicit constructions of expanders; see Theorem 2.4

of [27] for a proof. We identify the vertices of H with the vertices of G', so $V(H) = V'$. The running time of this step is $O(n')$. Using a simple greedy algorithm, and the fact that the maximum vertex degree in H is at most 9, we can partition the set $E(H)$ of edges into 17 matchings, $M_1, \ldots, M_{17}$. We then perform up to 17 iterations; in each iteration i, we will either embed the edges of M_i into G', after possibly adding a small number of fake edges to it, or we will compute the desired cut (A, B) in G'.

The ith iteration is executed as follows. We denote $M_i = \{e_1, \ldots, e_{k_i}\}$, where the edges are indexed in an arbitrary order. For each $1 \le j \le k_i$, denote $e_j = (u_j, v_j)$. We define two corresponding sets A_j, B_j of vertices of G', where $A_j = \{u_j\}$ and $B_j = \{v_j\}$. We then apply Algorithm RouteOrCut from Corollary 8.3.7 to graph G', the sets $A_1, B_2, \ldots, A_{k_i}, B_{k_i}$ of its vertices, integer $z = \left\lceil \frac{n'}{c \log^5 n'} \right\rceil$ for some large enough constant c, and parameter $\psi = 1/100$. Recall that the running time of the algorithm is $\tilde{O}(k_i |E(G')| \Delta^3/\psi^3 + k_i n'/\Delta^2) = \tilde{O}((n')^2)$. We now consider two cases. If the algorithm returns a cut (X, Y), with $\Psi_{G'}(X, Y) \le \psi$, then we terminate the algorithm and return this cut; in this case, $|X|, |Y| \ge z/2 \ge \Omega(n'/\log^5 n')$ must hold. Otherwise, the algorithm computes a partial routing $\mathcal{P}$ of the sets $A_1, B_1, \ldots, A_{k_i}, B_{k_i}$, of value at least $k_i - z$, that causes congestion at most $O(\Delta^2 \log^2 n'/\psi^2) = O(\log^4 n')$. Let $M_i' \subseteq M_i$ be the subset of edges that are routed in $\mathcal{P}$, so for every edge $e_j \in M_i'$ there is a path $P(e_j) \in \mathcal{P}$ connecting its endpoints. Let $M_i'' \subseteq M_i$ denote the set of the remaining edges, so $|M_i''| \le z$. We let $F_i = M_i''$ be a set of fake edges in graph G', that we use in order to route the edges of M_i''. For each edge $e_j \in M_i''$, we let $P(e_j)$ be the path consisting of the new fake copy of e_j in G'. Let $\mathcal{P}_i = \mathcal{P} \cup \{P(e_j) \mid e_j \in M_i''\}$. We now obtained an embedding of the edges of M_i into $G' + F_i$, with congestion $O(\log^4 n')$.

If the algorithm never terminates with the cut (X, Y) with $\Psi_{G'}(X, Y) \le \psi$, then, after 17 iterations, we obtain an embedding $\mathcal{P}^* = \bigcup_{i=1}^{17} \mathcal{P}_i$ of H into $G + F$, where $F = \bigcup_{i=1}^{17} F_i$ is a set of at most $17z$ fake edges; the congestion of the embedding is $\eta = O(\log^4 n')$. Moreover, if we denote by Δ_G the maximum vertex degree in $G + F$, then $\Delta_G \le 17 + \Delta \le 17\Delta$. Next, we apply Algorithm ExtractExpander from Lemma 8.2.6 to graphs G', H, the set F of fake edges, and the embedding $\mathcal{P}^*$ of H into G. Since $\frac{\psi^* n'}{32 \Delta_G \eta} \ge \frac{\psi^* n'}{O(\Delta \log^4 n')} \ge \frac{n'}{O(\log^5 n')}$, by letting the constant c used in the definition of z be large enough, we ensure that $|F| \le 17z \le \frac{\psi^* n'}{32 \Delta_G \eta}$, as required. The algorithm from Lemma 8.2.6 then computes a subgraph $G'' \subseteq G'$ that is a ψ'-expander, where $\psi' \ge \frac{\psi^*}{6 \Delta_G \eta} = \Omega\left(\frac{1}{\log^5 n'}\right)$, with:

$$V(G'') \ge n' - \frac{4 \cdot 17 z \eta}{\psi^*} = n' - O(z \log^4 n')$$

By letting c be a large enough constant, we can ensure that $|V(G'')| \ge 2n'/3$. The running time of Algorithm ExtractExpander from Lemma 8.2.6 is $\tilde{O}(|E(G')| \Delta_G \eta / \psi^*) = \tilde{O}(n')$, and so the total running time of the algorithm is $\tilde{O}((n')^2)$.

8.4.2 Step: $q > 1$

Suppose we are given an integer $q > 1$. We assume that Theorem 8.4.1 holds for $q-1$: that is, there is a deterministic algorithm, that we denote by $\mathcal{A}(q-1)$, that, given an n-vertex graph G with maximum vertex degree at most $c_{\text{CMG}} \log n$ and $n \leq N^{q-1}$, for some $N > N_0$, either returns a cut (A,B) in G with $|A|,|B| \geq n/4$ and $|E(A,B)| \leq n/100$, or it computes a subset $S \subseteq V(G)$ of at least $n/2$ vertices, such that $\Psi(G[S]) \geq \psi_{q-1}$, where $\psi_{q-1} = 1/\left((q-1)\log N\right)^{8(q-1)}$. We denote the running time of this algorithm by $T(q-1) = O\left(N^q \cdot ((q-1)\log N)^{c_0(q-1)^2}\right)$. Throughout the proof, we also denote $\psi_q = 1/\left(q\log N\right)^{8q}$

We now prove that the theorem holds for the given value of q, by invoking Algorithm $\mathcal{A}(q-1)$ a number of times. The following theorem is central to proving the induction step.

Theorem 8.4.3

There is a deterministic algorithm that, given as input an n'-vertex graph $G' = (V', E')$ and integers N, q with $N > N_0$ an integral power of 2 and $q > 1$, such that $N^{q-1}/2 \leq n' \leq N^q$, and maximum vertex degree of G' is $\Delta = O(\log n')$, returns one of the following:

- either a subset $S \subseteq V'$ of at least $2n'/3$ vertices such that $G'[S]$ is a ψ_q-expander; or
- a cut (X,Y) in G' with $|X|,|Y| \geq \Omega\left(\frac{\psi_{q-1}\cdot n'}{\log^8 n'}\right)$ and $\Psi_{G'}(X,Y) \leq 1/100$.

The running time of the algorithm is $O\left(N^{q+1}\cdot (q\log N)^{8q+O(1)}\right) + O\left(N \cdot \log n'\right)\cdot T(q-1)$.

We prove Theorem 8.4.3 below, after we complete the proof of Theorem 8.1.3 for the current value of q using it.

Note that we can assume that $n > N^{q-1}$, since otherwise we can use algorithm $\mathcal{A}(q-1)$, to either compute a cut (A,B) in G with $|A|,|B| \geq n/4$ and $|E(A,B)| \leq n/100$, or to compute a subset $S \subseteq V(G)$ of at least $n/2$ vertices, such that $\Psi(G[S]) \geq \psi_{q-1} \geq \psi_q$, in time $T(q-1) = O\left(N^q \cdot ((q-1)\log N)^{c_0(q-1)^2}\right)$.

Our algorithm performs a number of iterations. We maintain a subgraph $G' \subseteq G$; at the beginning of the algorithm, $G' = G$. In the ith iteration, we compute a subset $S_i \subseteq V(G')$ of vertices, and then update the graph G' by deleting the vertices of S_i from it. The iterations are performed as long as $|\bigcup_i S_i| < n/4$.

In order to execute the ith iteration, we consider the current graph G', denoting $|V(G')| = n'$. Note that, since we assume that $|\bigcup_{i'<i} S_{i'}| < n/4$, we get that $n' \geq 3n/4$, and in particular $N^{q-1}/2 \leq n' \leq N^q$. We then apply Theorem 8.4.3 to graph G'. If the outcome is a subset $S \subseteq V'$ of at least $2n'/3$ vertices such that $G'[S]$ is a ψ_q-expander, then we terminate the algorithm and return S. Notice that, since $n' \geq 3n/4$, and $|S| \geq 2n'/3$, we are guaranteed that $|S| \geq n/2$. Otherwise, we obtain a cut (X,Y) in G' with $|X|,|Y| \geq \Omega\left(\frac{\psi_{q-1}\cdot n'}{\log^8 n'}\right)$ and

$\Psi_{G'}(X,Y) \leq 1/100$. Assume w.l.o.g. that $|X| \leq |Y|$. We then set $S_i = X$, update the graph G' by removing the vertices of S_i from it, and continue to the next iteration. If the algorithm does not terminate with a ψ_q-expander, then it terminates once $|\bigcup_{i'} S_{i'}| \geq n/4$ holds. Let i denote the number of iterations in this case. Since we are guaranteed that $|\bigcup_{i'<i} S_{i'}| < n/4$, while $|S_i| \leq n'/2 \leq 3n'/8$, we get that $n/4 \leq |\bigcup_{i'=1}^{i} S_{i'}| \leq 5n/8$. Let $A = \bigcup_{i'=1}^{i} S_{i'}$, and let $B = V(G) \setminus A$. From the above discussion, we are guaranteed that $|A|, |B| \geq n/4$, and moreover, since the cut $S_{i'}$ that we obtain in every iteration i' has sparsity at most $1/100$ in its current graph G', it is easy to verify that $|E_G(A,B)| \leq |A|/100 \leq n/100$. We then return the cut (A,B) as the algorithm's outcome.

Notice that the number of iterations in the algorithm is bounded by:

$$O(\log^9 n/\psi_{q-1}) = O\left((q\log N)^{8(q-1)} \cdot \log^9 n\right) \leq O\left((q\log N)^{8q+1}\right),$$

since $n \leq N^q$. Therefore, the total running time of the algorithm is at most:

$$O\left(N^{q+1} \cdot (q\log N)^{16q+O(1)}\right) + O\left(N(q\log N)^{8q+2}\right) \cdot T(q-1).$$

From the induction hypothesis, $T(q-1) = O\left(N^q \cdot ((q-1)\log N)^{c_0(q-1)^2}\right)$. Assuming that $q \geq 1$, and that c_0 is a large enough constant, we get that the running time is $T(q) = O\left(N^{q+1} \cdot (q\log N)^{c_0 q^2}\right)$, as required.

In the remainder of this subsection we focus on the proof of Theorem 8.4.3.

Proof of Theorem 8.4.3

One of the main technical tools in the proof of the theorem is composition of expanders that we discuss next.

Composing Expanders. Suppose we are given a collection $\{G_1, \ldots, G_h\}$ of disjoint graphs, where for all $1 \leq i \leq h$, the set $V(G_i)$ of vertices, that is denoted by V_i, has cardinality at least N. Let H be another graph, whose vertex set is $\{v_1, \ldots, v_h\}$. An *N-composition* of H with $G_1, \ldots, G_h$ is another graph G, whose vertex set is $\bigcup_{i=1}^{h} V_i$, and whose edge set consists of two subsets: set $E^1 = \bigcup_{i=1}^{h} E(G_i)$, and another set E^2 of edges, defined as follows: for each edge $e = (v_i, v_j) \in E(H)$, let $M(e)$ be an arbitrary matching of cardinality N between vertices of V_i and vertices of V_j. Then $E^2 = \bigcup_{e \in E(H)} M(e)$. The following theorem shows that, if each of the graphs $G_1, \ldots, G_h$ is a ψ-expander, and graph H is a ψ'-expander, then the resulting graph G is also an expander for an appropriately chosen expansion parameter.

Theorem 8.4.4

Let $G_1, \ldots, G_h$ be a collection of $h > 1$ graphs, such that for each $1 \le i \le h$, $N \le |V(G_i)| \le \gamma N$, and G_i is a ψ-expander, for some $N \ge 1$, $\gamma \ge 1$, and $0 < \psi \le 1$. Let H be another graph with vertex set $\{v_1, \ldots, v_h\}$, such that H is a ψ'-expander, and let Δ be maximum vertex degree in H. Lastly, let G be a graph that is an N-composition of H with $G_1, \ldots, G_h$. Then graph G is a ψ''-expander, for $\psi'' = \psi\psi'/(16\Delta\gamma^2)$.

Proof. For convenience, for all $1 \le i \le h$, we denote $V(G_i)$ by V_i. Let (A, B) be any partition of $V(G)$. It is sufficient to prove that $|E_G(A, B)| \ge \psi'' \cdot \min\{|A|, |B|\}$.

Consider any graph G_i, for $1 \le i \le h$. We say that G_i is of type 1 if $|V_i \cap A| > \left(1 - \frac{1}{2\gamma}\right)|V_i|$, and we say that it is of type 2 if $|V_i \cap B| > \left(1 - \frac{1}{2\gamma}\right)|V_i|$. Notice that a graph G_i cannot belong to both types simultaneously, and it is possible that it does not belong to either type. Let N_1 be the number of type-1 graphs G_i, and let N_2 be the number of type-2 graphs. Assume w.l.o.g. that $N_1 \le N_2$. Let $S \subseteq V(H)$ contain all vertices v_i, such that G_i is a type-1 graph, so $|S| = N_1$. Since graph H is a ψ'-expander, $|E_H(S, V(H) \setminus S)| \ge \psi'|S|$.

We partition the set A of vertices into two subsets: set A' contains all vertices that lie in type-1 graphs G_i, and set A'' contains all remaining vertices. Recall that graph G contains, for every edge $e = (v_i, v_j) \in E_H(S, V(H) \setminus S)$, a collection $M(e)$ of N edges, connecting vertices of V_i to vertices of V_j. Consider any such edge $e = (v_i, v_j)$, with $v_i \in S$. Since $|V_i \cap A| \ge \left(1 - \frac{1}{2\gamma}\right)|V_i|$, and $|V_i| \le \gamma N$, $|V_i \cap B| \le \frac{|V_i|}{2\gamma} \le \frac{N}{2}$. Therefore, at least $N/2$ edges of $M(e)$ have one endpoint in A'; the other endpoint of each such edge must lie in $A'' \cup B$. We conclude that $|E_G(A', A'' \cup B)| \ge \frac{N \cdot |E_H(S, V(H) \setminus S)|}{2} \ge \frac{\psi' N |S|}{2}$. Since every graph G_i contains between N and γN vertices, we get that $|A'| \le \gamma N |S|$, and so $|E_G(A', A'' \cup B)| \ge \frac{\psi'|A'|}{2\gamma}$. Since maximum vertex degree in H is Δ, every vertex in A'' may be an endpoint of at most Δ such edges.

We now consider two cases. First, if $|A''| \le \psi'|A'|/(4\Delta\gamma)$, then $|E_G(A', A'')| \le \Delta|A''| \le \psi'|A'|/(4\gamma)$. Therefore, $|E_G(A, B)| \ge |E_G(A', B)| \ge \psi'|A'|/(4\gamma) \ge \psi'|A|/(8\gamma) \ge \psi''|A|$.

Lastly, assume that $|A''| > \psi'|A'|/(4\Delta\gamma)$, so $|A''| \ge \psi'|A|/(8\Delta\gamma)$. Consider any graph G_i that is not a type-1 graph, so $|V_i \cap A| \le \left(1 - \frac{1}{2\gamma}\right)|V_i|$. If $|V_i \cap A| \le |V_i|/2$, then there are at least $\psi|V_i \cap A|$ edges of G_i in $E_G(A, B)$. Otherwise, there are at least $\psi|V_i \cap B|$ edges of G_i in $E_G(A, B)$. Since $|V_i \cap B| \ge |V_i|/2\gamma \ge |V_i \cap A|/(2\gamma)$, the number of edges that G_i contributes to $E_G(A, B)$ is at least $\psi|V_i \cap B| \ge \psi|V_i \cap A|/\gamma$. We conclude that $|E_G(A, B)| \ge \psi|A''|/(2\gamma) \ge \psi\psi'|A|/(16\Delta\gamma^2) \ge \psi''|A|$. □

Proof Overview. We now provide an overview of the proof of Theorem 8.4.3, and set up some notation.

In order to simplify the notation, we denote the input graph by $G = (V, E)$, and we denote $|V| = n$ and $|E| = m$; recall that $|E| = O(n \log n)$. Let $\tilde{N}' = N^{q-1}/2$, and let $\tilde{N} = \left\lfloor n/\tilde{N}' \right\rfloor$, so $\tilde{N} \le 2N$. Since N is an integral power of 2, $\tilde{N}'$ is an even integer. Moreover, from our assumption that $n \geq N^{q-1}/2$, we get that $\tilde{N} \geq 1$.

We partition the set V of vertices into $\tilde{N}+1$ subsets $V_1, \ldots, V_{\tilde{N}}, V_{\tilde{N}+1}$, where sets $V_1, \ldots, V_{\tilde{N}}$ have cardinality exactly $\tilde{N}'$ each, and the last set, that we denote by $Z = V_{\tilde{N}+1}$ has cardinality less than $\tilde{N}'$. We call the vertices in Z the *extra vertices*.

The algorithm consists of three steps. In the first step, we construct expanders $H_1, \ldots, H_{\tilde{N}}$, where for all $1 \leq i \leq \tilde{N}$, $V(H_i) = V_i$, that we attempt to embed into G. We will either succeed in embedding these expanders with a small congestion and a relatively small number of fake edges, or we will compute the desired cut (X, Y) in G. In the second step, we construct an expander H' whose vertex set is $v_1, \ldots, v_{\tilde{N}}$, where we think of vertex v_i as representing the set V_i of vertices of G. We will attempt to embed graph H' into G, with a small number of fake edges and low congestion, where every edge $e = (v_i, v_j)$ of H' is embedded into $\tilde{N}'$ paths connecting vertices of V_i to vertices of V_j in G. If our algorithm fails to find such an embedding, then we will again produce the desired cut (X, Y) in G. If, over the course of the first two steps, the algorithm does not terminate with a cut (X, Y) in G, then we consider an expander H^*, obtained by computing a $\tilde{N}'$-composition of $H_1, \ldots, H_N$ and of H', and then adding the vertices of Z, together with a matching connecting every vertex of Z to some vertex of $V_1 \cup \cdots \cup V_{\tilde{N}}$ to the resulting graph. The algorithm from the first two steps has then computed an embedding of H^* into G, with a relatively small number of fake edges. In our last step, we compute a large subset S of vertices of G such that $G[S]$ is a ψ_q-expander, using Algorithm ExtractExpander from Lemma 8.2.6. We now proceed to describe each of the three steps in turn. Throughout the algorithm, we use a parameter $z = \frac{\psi_{q-1} n}{c \log^8 n}$, where c is a large enough constant, whose value will be set later.

Step 1: Embedding Many Small Expanders. The goal of this step is to construct a collection $\mathcal{H} = \{H_1, \ldots, H_{\tilde{N}}\}$ of expanders, where for $1 \leq i \leq \tilde{N}$, $V(H_i) = V_i$, and to compute an low-congestion embedding of all these expanders into $G + F$, where F is a small set of fake edges for G. In other words, if we let H be the graph obtained by taking a disjoint union of the graphs $H_1, \ldots, H_{\tilde{N}}$, and the set Z of isolated vertices, then we will attempt to compute an embedding of H into G. We will either find such an embedding, that uses relatively few fake edges, or we will return a cut (X, Y) of G with the required properties. We summarize this step in the following lemma.

Lemma 8.4.5

There is a deterministic algorithm that either computes a cut (X, Y) in G with $|X|, |Y| \geq \Omega\left(\frac{\psi_{q-1} \cdot n}{\log^8 n}\right)$ and $\Psi_G(X, Y) \leq 1/100$; or it constructs a collection $\mathcal{H} = \{H_1, \ldots, H_{\tilde{N}}\}$ of $\hat{\psi}$-expanders, where for $1 \leq i \leq N$, $V(H_i) = V_i$, and $\hat{\psi} = \psi_{q-1}/2$, together with a set F of at most $O(z \log n)$ fake edges, and an embedding $\mathcal{P}$ of the graph $H = (\bigcup_i H_i) \cup Z$ into $G + F$, with congestion $O(\log^5 n)$, such that every vertex of G is incident to at most $O(\log n)$ edges of F. The running time of the algorithm is $O\left(N^{q+1} \cdot \operatorname{poly}\log n\right) + O\left(N \cdot \log n\right) \cdot T(q-1)$.

Proof. The construction of the graphs $H_1, \ldots, H_{\tilde{N}}$, and of their embedding into G is done gradually, by running $\tilde{N}$ instances of the cut-matching game, in parallel. Initially, for each $1 \le i \le \tilde{N}$, we let the graph H_i contain the set V_i of vertices and no edges. Throughout the algorithm, we denote by $\mathcal{H} = \{H_1, \ldots, H_{\tilde{N}}\}$ the current collection of the expanders we are constructing. We partition $\mathcal{H}$ into two subsets: set $\mathcal{H}'$ of *active* graphs, and set $\mathcal{H}''$ of *inactive* graphs. Initially, every graph H_i is active, so $\mathcal{H}' = \mathcal{H}$ and $\mathcal{H}'' = \emptyset$. Throughout the algorithm, for every inactive graph H_i, we will maintain a subset $S_i \subseteq V_i$ of at least $\tilde{N}'/2$ vertices, such that graph $H_i[S_i]$ is a ψ_{q-1}-expander. Throughout the algorithm, we also let H denote the graph obtained by taking the disjoint union of all graphs in $\mathcal{H}$ with a set Z of isolated vertices. We will maintain an embedding $\mathcal{P}$ of H into G throughout the algorithm. We will ensure that, throughout the algorithm, for all $1 \le i \le \tilde{N}$, the maximum vertex degree in each graph H_i is at most $c_{\text{CMG}} \log \tilde{N}'$.

At the beginning of the algorithm, for each $1 \le i \le \tilde{N}$, graph H_i contains the set V_i of vertices and no edges, so graph H consists of the set V of vertices and no edges. The initial embedding is $\mathcal{P} = \emptyset$, and every graph H_i is active.

As long as $\mathcal{H}' \neq \emptyset$, we perform iterations, where the jth iteration is executed as follows. We apply algorithm $\mathcal{A}(q-1)$ to every graph $H_i \in \mathcal{H}'$ separately. Observe that each such graph contains $\tilde{N}' \le N^{q-1}$ vertices, and has maximum vertex degree at most $c_{\text{CMG}} \log \tilde{N}'$. For each such graph H_i, if the outcome is a subset $S_i \subseteq V_i$ of vertices, such that $|S_i| \ge \tilde{N}'/2$ and $H_i[S_i]$ is a ψ_{q-1}-expander, then we add H_i to the set $\mathcal{H}''$ of inactive graphs, and store the set S_i of vertices with it. Let $\hat{\mathcal{H}} \subseteq \mathcal{H}'$ be the collection of all remaining active graphs, so for each graph $H_i \in \hat{\mathcal{H}}$, the algorithm has computed a cut (A_i, B_i) with $|A_i|, |B_i| \ge \tilde{N}'/4$, and $|E_{H_i}(A_i, B_i)| \le \tilde{N}'/100$. We assume without loss of generality that $|A_i| \le |B_i|$. Let (A'_i, B'_i) be any partition of V_i with $|A'_i| = |B'_i|$, such that $A_i \subseteq A'_i$. We treat the partition (A'_i, B'_i) as the move of the cut player in the cut-matching game corresponding to the graph H_i.

For convenience, we assume w.l.o.g. that $\hat{\mathcal{H}} = \{H_1, \ldots, H_k\}$. In order to implement the response of the matching player, we apply Algorithm RouteOrCut from Corollary 8.3.7 to graph G, the sets $A'_1, B'_1, \ldots, A'_k, B'_k$ of vertices, and parameters $\psi = 1/100$ and z (recall that we have defined $z = \frac{\psi_{q-1} n}{c \log^8 n}$ for some large enough constant c). We now consider two cases. If Algorithm RouteOrCut from Corollary 8.3.7 returns a cut (X, Y) of G with $|X|, |Y| \ge z/2$ and $\Psi_G(X, Y) \le \psi$, then we say that the current iteration terminates with a cut. In this case, we terminate the algorithm, and return (X, Y) as its outcome; it is immediate to verify that this cut has the required properties. In the second case, we obtain a partial routing $(M' = \bigcup_{i=1}^{k} M'_i, \mathcal{P}')$ of the sets $A'_1, B'_1, \ldots, A'_k, B'_k$ of vertices, where $|M'| \ge k\tilde{N}'/2 - z$ (recall that for all i, $|A'_i| = |V_i|/2 = \tilde{N}'/2$). The congestion of the embbedding is at most $O(\Delta^2 \log^2 n / \psi^2) = O(\log^4 n)$. We then say that the current iteration has terminated with a routing.

Consider now some index $1 \le i \le k$, and let $A''_i \subseteq A'_i$ and $B''_i \subseteq B'_i$ be the subsets of vertices that do not participate in the matching M'_i. Let M''_i be an arbitrary perfect matching between the vertices of A''_i and the vertices of B''_i, and let F_i be a set of fake edges

$F_i = \{(u,v) \mid (u,v) \in M''_i\}$. For every pair $(u,v) \in M''_i$, we embed the pair (u,v) into the corresponding fake edge $(u,v) \in F_i$. Let $M^j_i = M'_i \cup M''_i$. We add the edges of M^j_i to graph H_i.

Denote $M^j = \bigcup_{i=1}^{k} M^j_i$, and let $F^j = \bigcup_{i=1}^{k} F_i$ be the resulting set of fake edges; recall that $|F^j| \leq z$. Let $\mathcal{P}^j$ be the embedding of all edges in M^j that is obtained from the partial routing $\mathcal{P}'$, by adding the embeddings of all fake edges to it. Observe that we have now obtained an embedding $\mathcal{P}^j$ of all edges of M^j into $G + F^j$, with congestion $O(\log^4 n)$. We add the paths of $\mathcal{P}^j$ to the embedding $\mathcal{P}$ of the current graph H, and continue to the next iteration.

Our algorithm can therefore be viewed as running $\tilde{N}$ parallel copies of the cut-matching game. From Theorem 8.2.2, the number of iterations is bounded by $c_{\text{CMG}} \log \tilde{N}'$, and so for every graph H_i, its maximum vertex degree is always bounded by $c_{\text{CMG}} \log \tilde{N}'$. The algorithm terminates once all graphs H_i become inactive. Recall that for each such graph H_i, we are given a subset S_i of its vertices, such that $|S_i| \geq |V_i|/2$, and $H_i[S_i]$ is a ψ_{q-1}-expander. We perform one last iteration, whose goal is to turn each graph H_i into an expander, by adding a new set of edges to it, while simultaneously embedding these edges into the graph G together with a small number of fake edges, or find a cut (X,Y) as required. Let $r-1$ denote the index of the last iteration before every graph H_i becomes inactive.

Last Iteration. For each $1 \leq i \leq \tilde{N}$, we let $B_i = S_i$ and $A_i = V_i \setminus S_i$, so that $|A_i| \leq |B_i|$ holds. We apply Algorithm RouteOrCut from Corollary 8.3.7 to graph G, the sets $A_1, B_1, \ldots, A_{\tilde{N}}, B_{\tilde{N}}$ of vertices, and parameters $\psi = 1/100$ and z. The remainder of the iteration is executed exactly as before. If Algorithm RouteOrCut from Corollary 8.3.7 returns a cut (X,Y) of G with $|X|, |Y| \geq z/2$ and $\Psi_G(X,Y) \leq \psi$, then we terminate the algorithm and return this cut. Otherwise, we obtain a partial routing $(M' = \bigcup_{i=1}^{\tilde{N}} M'_i, \mathcal{P}')$ of the sets $A_1, B_1, \ldots, A_{\tilde{N}}, B_{\tilde{N}}$ of vertices, where $|M'| \geq \sum_{i=1}^{\tilde{N}} |A_i| - z$, whose congestion is at most $O(\log^4 n)$ as before.

Consider some index $1 \leq i \leq \tilde{N}$, and let $A'_i \subseteq A_i$ and $B'_i \subseteq B_i$ be the subsets of vertices that do not participate in the matching M'_i. Let M''_i be an arbitrary matching, in which every vertex of A'_i is matched to some vertex of B'_i, and let F_i be a set of fake edges corresponding to this matching M''_i, defined as before. For every pair $e = (u,v) \in M''_i$, we embed the fake edge e into the path $P(e) = (e)$. Let $M^r_i = M'_i \cup M''_i$. We add the edges of M^r_i to graph H_i.

Denote $M^r = \bigcup_{i=1}^{\tilde{N}} M^r_i$, and let $F^r = \bigcup_{i=1}^{\tilde{N}} F_i$ be the resulting set of fake edges; as before $|F^r| \leq z$. Let $\mathcal{P}^r$ be the embedding of all edges in M^r that is obtained from the partial routing $\mathcal{P}'$, by adding the embeddings of all fake edges to it. As before, we have obtained an embedding $\mathcal{P}^r$ of all edges of M^r into $G + F^r$, with congestion $O(\log^4 n)$. We add the paths of $\mathcal{P}^r$ to the embedding $\mathcal{P}$ of the current graph H. It is not hard to see that every graph $H_i \in \mathcal{H}$ is now a $\psi_{q-1}/2$-expander.

Recall that the congestion incurred by each path set $\mathcal{P}^j$ of edges is $O(\log^4 n)$, and, since the number of iterations is $O(\log n)$, the embedding $\mathcal{P}$ causes congestion $O(\log^5 n)$. The total number of fake edges in $F = \bigcup_{j=1}^{r} F^j$ is $O(z \log n)$. Since each set F^j of fake edges is a matching, every vertex of G is incident to $O(\log n)$ fake edges.

We now analyze the running time of the algorithm. As observed before, the algorithm has $O(\log n)$ iterations. In every iteration, we apply algorithm $\mathcal{A}(q-1)$ to $\tilde{N} = O(N)$ graphs. Additionally, we use Algorithm RouteOrCut from Corollary 8.3.7, whose running time is $\tilde{O}(k|E(G)|/\psi^3 + kn/\psi^2) = \tilde{O}(kn)$, where k is the number of vertex subsets. Since $k \leq |\mathcal{H}| = \tilde{N} \leq N$, this running time is bounded by $\tilde{O}(Nn) = \tilde{O}(N^{q+1})$. Therefore, the total running time of the algorithm is $\tilde{O}\left(N^{q+1}\right) + O\left(N \log n\right) \cdot T(q-1)$.

□

Step 2: Embedding One Large Expander. We use Lemma 8.2.1 to construct, in time $O(\tilde{N})$, a ψ^*-expander $H' = H_{\tilde{N}}$ on $\tilde{N}$ vertices, with $\psi^* = \Psi(H') = \Omega(1)$, such that maximum vertex degree in H' is at most 9. For convenience, we denote $V(H') = \{v_1, \ldots, v_{\tilde{N}}\}$. The main part of this step is summarized in the following lemma.

Lemma 8.4.6

There is a deterministic algorithm, that either computes a cut (X, Y) in G with $|X|, |Y| \geq \Omega\left(\frac{\psi_{q-1} \cdot n}{\log^8 n}\right)$ and $\Psi_G(X, Y) \leq 1/100$; or it computes a collection F' of at most $17z$ fake edges in G, and, for every edge $e = (v_i, v_j) \in E(H')$ a set $\mathcal{P}(e)$ of $\tilde{N}'$ paths in $G + F'$, such that every path in $\mathcal{P}(e)$ connects a vertex of V_i to a vertex of V_j, and the endpoints of the paths in $\mathcal{P}(e)$ are disjoint. Moreover, every vertex of G is incident to at most 17 fake edges in F', and every edge of $G \cup F'$ participates in at most $O(\log^4 n)$ paths in $\bigcup_{e \in E(H')} \mathcal{P}(e)$. The running time of the algorithm is $\tilde{O}\left(N^{q+1}\right)$.

Proof. Using a standard greedy algorithm, and the fact that the maximum vertex degree in H' is at most 9, we can partition the set $E(H')$ of edges into 17 matchings, $M_1, \ldots, M_{17}$. We then perform up to 17 iterations; in each iteration i, we will either compute a small set F^i of fake edges for G, and the sets $\mathcal{P}(e)$ of paths for all edges $e \in M_i$, in graph $G + F^i$, or we will compute the cut (X, Y) in G with the required properties.

In order to execute the ith iteration, we consider the set M_i of edges of H', and denote, for convenience, $M_i = \{e_1, \ldots, e_{k_i}\}$. For each $1 \leq j \leq k_i$, if $e_j = (v_z, v_{z'})$, then we define $A_j = V_z$ and $B_j = V_{z'}$. Observe that $|A_i| = |B_j| = \tilde{N}'$, and the resulting vertex sets $A_1, B_1, \ldots, A_{k_i}, B_{k_i}$ are all disjoint.

We apply Algorithm RouteOrCut from Corollary 8.3.7 to graph G, the sets $A_1, B_1, \ldots, A_{k_i}, B_{k_i}$ of vertices, and parameters $\psi = 1/100$ and z (as defined before, $z = \frac{\psi_{q-1} n}{c \log^8 n}$). If Algorithm RouteOrCut returns a cut (X, Y) of G with $|X|, |Y| \geq z/2$ and $\Psi_G(X, Y) \leq \psi$, then we terminate the algorithm and return this cut; it is easy to verify that cut (X, Y) has all required properties. In this case we say that the iteration terminates with a cut. Otherwise, we obtain a partial routing $(\hat{M}_i = \bigcup_{e \in M_i} \hat{M}(e), \hat{\mathcal{P}}_i)$ of the sets $A_1, B_1, \ldots, A_{k_i}, B_{k_i}$ of vertices, where $|\hat{M}_i| \geq \sum_{j=1}^{k_i} |A_j| - z$, whose congestion is at most

$O(\Delta^2 \log^2 n/\psi^2) = O(\log^4 n)$. In this case we say that the iteration terminates with a routing. Consider now some edge $e_j \in M_i$. Let $A'_j \subseteq A_j$, $B'_j \subseteq B_j$ be the subsets of vertices that do not participate in the matching $\hat{M}(e_j)$. Let $\hat{M}'(e_j)$ be an arbitrary perfect matching between A'_j and B'_j, and let F^i_j be the corresponding set of fake edges for graph G (so for every edge $e \in \hat{M}'(e_j)$, we add an edge with the same endpoints to F^i_j). Finally, set $\hat{M}''(e_j) = \hat{M}_j \cup \hat{M}'_j$. Let $F^i = \bigcup_{j=1}^{k_i} F^i_j$; recall that $|F^i| \le z$. Let $\mathcal{P}'_i$ be the set of paths routing the edges of F^i, where for each edge $e \in F^i$, the corresponding path $P(e) \in \mathcal{P}'_i$ consists of the edge e. Lastly, let $\hat{\mathcal{P}}''_i = \hat{\mathcal{P}}_i \cup \hat{\mathcal{P}}'_i$. Note that $\hat{\mathcal{P}}''_i$ is the routing of all edges in $\hat{M}''_i \cup F^i$ in graph $G + F^i$, that causes edge-congestion at most $O(\log^4 n)$.

If any iteration of the algorithm terminated with a cut, then we terminate the algorithm and return the corresponding cut. We assume from now on that every iteration of the algorithm terminated with a routing. Setting $F' = \bigcup_{i=1}^{17} F^i$, we obtain the desired routing of the edges of H' in graph $G + F'$, with congestion $O(\log^4 n)$. Since, for every $1 \le i \le 17$, the edges of F^i form a matching, every vertex of G is incident to at most 17 such edges.

Recall that the running time of Algorithm RouteOrCut from Corollary 8.3.7 is $\tilde{O}(k|E(G)|/\psi^3 + kn/\psi^2) = \tilde{O}(kn) = \tilde{O}(kN^q)$, where k is the number of pairs of sets that we need to route. Since $k \le |V(H')| \le \tilde{N} \le O(N)$, and the number of iterations is at most 17, we get that the running time of the algorithm is $\tilde{O}\left(N^{q+1}\right)$. □

Finally, we need the following claim, in order to connect the set Z of extra vertices to the remaining vertices of G.

Claim 8.4.7

There is a deterministic algorithm, that either computes a cut (X, Y) in G with $|X|, |Y| \ge \Omega\left(\frac{\psi_{q-1} \cdot n}{\log^8 n}\right)$ and $\Psi_G(X, Y) \le 1/100$; or it computes a matching M connecting every vertex of Z to a distinct vertex of $V(G) \setminus Z$, a collection F'' of at most z fake edges in G, and a set $\mathcal{P}'' = \{P(e) \mid e \in M\}$ of paths in $G + F''$, such that, for each edge $e = (u, v) \in M$, path $P(e)$ connects u to v. Moreover, every vertex of G is incident to at most one fake edge in F'', and every edge of $G \cup F''$ participates in at most $O(\log^4 n)$ paths in $\mathcal{P}''$. The running time of the algorithm is $\tilde{O}\left(N^q\right)$.

Proof. We apply Algorithm RouteOrCut from Corollary 8.3.7 to graph G, the sets $A_1 = Z$, $B_1 = V(G) \setminus Z$ of vertices, parameter $\psi = 1/100$, and parameter z. If the outcome of Algorithm RouteOrCut is a cut (X, Y) of G with $|X|, |Y| \ge z/2$ and $\Psi_G(X, Y) \le \psi$, then we return this cut; it is immediate to verify that cut (X, Y) has the required properties. Otherwise, we obtain a routing $(M', \mathcal{P}')$ of the sets A_1, B_1, with $|M'| \ge |Z| - z$. The congestion of the routing is at most $O(\Delta^2 \log^2 n/\psi^2) = O(\log^4 n)$. We let $Z' \subseteq Z$ be the set of all vertices of Z that do not participate in the matching M', and we let M'' be an arbitrary matching that matches every vertex of Z' to a distinct vertex of $V(G) \setminus Z$, such that $M = M' \cup M''$ is a matching; such a set M'' exists since Z contains at most half the vertices of G. We let F'' be a set of fake edges for G corresponding to the edges of M'', so every edge $e = (u, v) \in M''$ is also added to F''. We let $P(e)$ be the path that only consists of the edge e, and we treat $P(e)$ as the embedding of e.

We now obtained a set F'' of at most z fake edges, and every vertex of G is incident to at most one such fake edge. We also obtained an embedding $\mathcal{P}'' = \mathcal{P}' \cup \{P(e) \mid e \in F''\}$ of M into G with congestion $O(\log^4 n)$. The running time of Algorithm RouteOrCut is $\tilde{O}(\Delta^3|E(G)|/\psi^3 + n/\psi^2) = \tilde{O}(n) = \tilde{O}(N^q)$. □

If the algorithm from Lemma 8.4.6 or the algorithm from Claim 8.4.7 produce a cut (X, Y) in G with $|X|, |Y| \geq \Omega\left(\frac{\psi_{q-1}\cdot n}{\log^8 n}\right)$ and $\Psi_G(X, Y) \leq 1/100$, then we terminate the algorithm and return this cut. Otherwise, consider the following graph H^*: we start by letting H^* be a disjoint union of the graphs $H_1, \ldots, H_{\tilde{N}}$ constructed in the first step. Additionally, for every edge $e = (v_i, v_j) \in E(H')$, for every path $P \in \mathcal{P}(e)$, whose endpoints are $x \in V_i$, $y \in V_j$, we add the edge (x, y) to $E(H^*)$. It is immediate to verify that graph H^* is an N'-composition of $H_1, \ldots, H_{\tilde{N}}$, and graph H'. Recall that for all $1 \leq i \leq N$, graph H_i is a $\psi_{q-1}/2$-expander, while graph H' is a ψ^*-expander, for some $\psi^* = \Omega(1)$. The maximum vertex degree in H' is bounded by 9. Therefore, from Theorem 8.4.4, graph H^* is a ψ'-expander, for $\psi' = \psi_{q-1}\psi^*/O(\log n) = \Omega(\psi_{q-1}/\log n)$. Note that the maximum vertex degree in H^* is $O(\log n)$. Lastly, we add to graph H^* the set Z of extra vertices as isolated vertices, and the matching M that was computed in Claim 8.4.7. Observe that graph H^* is a $\psi'/2$-expander, where $\psi' = \Omega(\psi_{q-1}/\log n)$; to simplify the notation, we say that H^* is a ψ'-expander, adjusting the value of ψ' accordingly. Let $F^* = F \cup F' \cup F''$ be the union of the sets of fake edges computed by the algorithms from Lemma 8.4.5, Lemma 8.4.6, and Claim 8.4.7. Recall that $|F^*| = O(z \log n)$, where $z = \frac{\psi_{q-1}n}{c\log^8 n}$ for some large enough constant c.

We denote by Δ_G the maximum vertex degree of $G + F^*$. Since the set F^* of fake edges consists of $O(\log n)$ matchings, $\Delta_G = O(\log n)$.

By combining the outcomes of the algorithms from Lemma 8.4.5, Lemma 8.4.6, and Claim 8.4.7, we obtain an embedding of H^* into $G + F^*$ with congestion at most $O(\log^5 n)$. The maximum vertex degree in H^*, that we denote by Δ_{H^*}, is $O(\log n)$. The maximum vertex degree in $G + F^*$, that we denote by Δ_G, is $O(\log n)$. Note that the running time of the algorithm so far is $O\left(N^{q+1} \cdot \text{poly}\log n\right) + O\left(N \cdot \log n\right) \cdot T(q-1)$.

Step 3: Obtaining the Final Expander. In this step, we apply Algorithm ExtractExpander from Lemma 8.2.6 to graphs G and H^*, the set F^* of fake edges, and the embedding of H^* into $G + F^*$ with congestion at most $\eta = O(\log^5 n)$. We need first to verify that $|F^*| \leq \frac{\psi' n}{32\Delta_G \eta}$. Recall that $\psi' = \Omega(\psi_{q-1}/\log n)$, $\Delta_G = O(\log n)$, and $\eta = O(\log^5 n)$. Therefore, $\frac{\psi' n}{32\Delta_G \eta} \geq \Omega\left(\frac{\psi_{q-1} n}{\log^7 n}\right)$, while $|F^*| \leq O(\log n) \cdot \frac{\psi_{q-1} n}{c\log^8 n}$. Setting the constant c to be large enough, we can ensure that the inequality indeed holds.

Recall that Algorithm ExtractExpander from Lemma 8.2.6 computes a subgraph $G' \subseteq G$, that is a ψ''-expander, for $\psi'' \geq \frac{\psi'}{6\Delta_G \cdot \eta} = \Omega\left(\frac{\psi_{q-1}}{\log^7 n}\right)$, as $\psi' = \Omega(\psi_{q-1}/\log n)$. Recall also that $\psi_{q-1} = 1/((q-1)\log N)^{8(q-1)}$, and $n \le N^q$. Therefore:

$$\psi'' \ge \Omega\left(\frac{1}{((q-1)\log N)^{8(q-1)} \cdot (q\log N)^7}\right) \ge \frac{1}{(q\log N)^{8q}} = \psi_q.$$

Note that the number of vertices in G' is at least: $n - \frac{4|F^*|\eta}{\psi'}$. Since $\frac{4|F^*|\eta}{\psi'} \le O\left(\frac{z\log^7 n}{\psi_{q-1}}\right)$ and $z = \frac{\psi_{q-1}n}{c\log^8 n}$, letting c be a large enough constant, we can ensure that $|V(G')| \ge 2n/3$, as required.

The running time of Algorithm ExtractExpander is $\tilde{O}(|E(G)|\Delta_G \cdot \eta/\psi') = \tilde{O}(n/\psi_{q-1})$ $= \tilde{O}(N^q \cdot (q\log N)^{8q})$.

By combining all three steps together, we obtain total running time $O\left(N^{q+1} \cdot (q\log N)^{8q+O(1)}\right) + O(N \cdot \log n) \cdot T(q-1)$, as required.

8.5 A Slower Algorithm for BalCutPrune

In this section we prove the following:

Theorem 8.5.1

There is a universal constant c, and a deterministic algorithm, that, given an n-vertex m-edge graph $G = (V, E)$, a parameter $0 < \phi < 1$, and another parameter $r \le c\log m$, returns a cut (A, B) in G with $|E_G(A, B)| \le \phi \cdot \mathbf{vol}(G)$, such that:

- either $\mathbf{vol}_G(A), \mathbf{vol}_G(B) \ge \mathbf{vol}(G)/3$; or
- $\mathbf{vol}_G(A) \ge \frac{7}{12} \cdot \mathbf{vol}(G)$, and the graph $G[A]$ has conductance $\phi' \ge \phi/\log^{O(r)} m$.

The running time of the algorithm is $O\left(m^{1+O(1/r)} \cdot (\log m)^{O(r^2)}/\phi^2\right)$.

From the definition of the BalCutPrune problem from Definition 8.1.1, this implies a slower version of Theorem 8.1.2 when the conductance parameter ϕ is low:

Corollary 8.5.2: Deterministic BalCutPrune algorithm

There is a deterministic algorithm, that, given a graph G with m edges, and parameters $\phi \in (0, 1]$, $1 \le r \le O(\log m)$, and $\alpha = (\log m)^{O(r)}$, computes an α-approximate solution to instance (G, ϕ) of BalCutPrune in time $O\left(m^{1+O(1/r)} \cdot (\log m)^{O(r^2)}/\phi^2\right)$.

While the above algorithm can significantly slower than the one from Theorem 8.1.2 when the conductance parameter ϕ is low, many of our applications only need to solve the BalCutPrune problem for relatively high values of ϕ, and so the algorithm from Theorem 8.5.1 is sufficient for them. In particular, we will use this algorithm in order to obtain fast deterministic approximation algorithms for max s-t flow, which will then in turn be used in order to obtain the full proof of Theorem 8.1.2. The remainder of this section is dedicated to the proof of Theorem 8.5.1.

Two key ingredients in the proof are an extension of Theorem 8.1.3 to smaller sparsity regime, and a degree reduction procedure, that are discussed in the next two subsections, respectively.

8.5.1 Extension of Theorem 8.1.3 to Smaller Sparsity

The algorithm from Theorem 8.1.3 only guarantees to find a cut with sparsity at most 1/25. In this subsection, we show an extension of Theorem 8.1.3 that is given a target sparsity parameter ψ (which can be much smaller than 1/25), and if the algorithm returns a cut, then that cut has sparsity at most ψ:

Lemma 8.5.3

There is a deterministic algorithm, that, given an n-vertex graph $G = (V, E)$, with maximum vertex degree Δ, parameters $0 < \psi < 1$, $z \geq 0$ and $r \geq 1$, such that $n^{1/r} \geq N_0$ (where N_0 is the constant from Theorem 8.1.3), returns one of the following:

- either a cut (X, Y) in G with $|X|, |Y| \geq z/\Delta$ and $\Psi_G(X, Y) \leq \psi$; or
- a graph H with $V(H) = V(G)$, that is a $\psi_r(n)$-expander (for $\psi_r(n) = 1/(\log n)^{O(r)}$), together with a set F of at most $O(z \log n)$ fake edges for G, and an embedding of H into $G + F$ with congestion at most $O(\Delta \log n/\psi)$, such that every vertex of G is incident to at most $O(\log n)$ edges of F.

The running time of the algorithm is $\tilde{O}\left(n^{1+O(1/r)} \cdot (\log n)^{O(r^2)} + n\Delta^2/\psi\right)$.

Proof. If the number of vertices in graph G is odd, then we add an additional new vertex v_0, and we connect it to an arbitrary vertex of G with a fake edge. For simplicity, the new number of vertices is still denoted by n.

Our algorithm runs the cut-matching game, as follows. We start with a graph H, whose vertex set is V, and whose edge set is empty, and then perform iterations. Throughout the algorithm, we will ensure that the maximum vertex degree in H is $O(\log n)$.

Iteration i is executed as follows. We apply Algorithm CutOrCertify from Theorem 8.1.3 to graph H. We now consider two cases. In the first case, the outcome is a cut (A_i, B_i) in H, with $|A_i|, |B_i| \geq n/4$ and $|E_H(A_i, B_i)| \leq n/100$. Let (A'_i, B'_i) be any partition of V with $A_i \subseteq A'_i$, $B_i \subseteq B'_i$, and $|A'_i| = |B'_i|$. We apply Algorithm RouteOrCut-1Pair from Theorem 8.3.8 to graph G, with the vertex sets A'_i, B'_i, and parameters z and ψ. If the outcome is a cut (X, Y) in G with $|X|, |Y| \geq z/\Delta$ and $\Psi_G(X, Y) \leq \psi$, then we terminate the algorithm and return this cut as its outcome. Otherwise, we obtain a partial routing $(M_i, \mathcal{P}_i)$ of the sets A'_i, B'_i, of value at least $|A'_i| - z$, that causes congestion at most $4\Delta/\psi$. Let $A''_i \subseteq A'_i$, $B''_i \subseteq B'_i$ be subsets of vertices that do not participate in the matching M_i. Let M'_i be an arbitrary perfect matching between A''_i and B''_i, and let F_i be a set of fake edges corresponding to the matching M'_i (so every edge in the matching becomes a fake edge). For every edge $e \in F_i$, we also let $P(e)$ be a path consisting of only the fake edge e. Let

$M''_i = M_i \cup M'_i$, and let $\mathcal{P}'_i = \mathcal{P}_i \cup \{P(e) \mid e \in F_i\}$. Then M''_i is a perfect matching between A'_i and B'_i, and $\mathcal{P}'_i$ is a routing of this matching in $G \cup F_i$, with congestion at most $4\Delta/\psi$. We add the edges of M''_i to H, and continue to the next iteration.

Consider now the second case, where the outcome of Algorithm CutOrCertify from Theorem 8.1.3 is a subset $S \subseteq V$ of at least $n/2$ vertices, such that $\Psi(G[S]) \geq \psi_r(n)$. Let i^* be the index of the current iteration. We then let $B_{i^*} = S$ and $A_{i^*} = V \setminus S$; note that $|A_{i^*}| \leq |B_{i^*}|$ must hold. We again employ Algorithm RouteOrCut-1Pair from Theorem 8.3.8, with the vertex sets A_{i^*}, B_{i^*}, and parameters z and ψ. If the outcome is a cut (X, Y) in G with $|X|, |Y| \geq z/\Delta$ and $\Psi_G(X, Y) \leq \psi$, then we terminate the algorithm and return this cut as its outcome. Otherwise, we obtain a partial routing $(M_{i^*}, \mathcal{P}_{i^*})$ of the sets A_{i^*}, B_{i^*}, of value at least $|A_{i^*}| - z$, that causes congestion at most $4\Delta/\psi$. As before, we let $A'_{i^*} \subseteq A_{i^*}$, $B'_{i^*} \subseteq B_{i^*}$ be subsets of vertices that do not participate in the matching M_{i^*}. Let M'_{i^*} be an arbitrary matching, that matches every vertex of A'_{i^*} to some vertex of B'_{i^*}, and let F_{i^*} be a set of fake edges corresponding to the matching M'_{i^*}. As before, for every edge $e \in F_{i^*}$, we let $P(e)$ be a path consisting of only the fake edge e. Let $M''_{i^*} = M_{i^*} \cup M'_{i^*}$, and let $\mathcal{P}'_{i^*} = \mathcal{P}_{i^*} \cup \{P(e) \mid e \in F_{i^*}\}$. Then M''_{i^*} matches every vertex of A_{i^*} to a distinct vertex of B_{i^*}, and $\mathcal{P}'_{i^*}$ is a routing of this matching in $G \cup F_{i^*}$, with congestion at most $4\Delta/\psi$. We add the edges of M''_{i^*} to H, and terminate the algorithm.

Observe that, if the algorithm never terminates with a cut (X, Y) with $|X|, |Y| \geq z/\Delta$ and $\Psi_{G'}(X, Y) \leq \psi$, then the final graph H is a $\psi_r(n)/2$-expander. Moreover, if we let $F = \bigcup_{i=1}^{i^*} F_i$, together with an additional fake edge incident to v_0 if the initial number of vertices in G was odd, and $\mathcal{P} = \bigcup_{i=1}^{i^*} \mathcal{P}'_i$, then $\mathcal{P}$ is an embedding of H into $G + F$. From Theorem 8.2.2, the number of iterations in the algorithm is bounded by $O(\log n)$. Since, for all i, edge set F_i is a matching, every vertex of G is incident to $O(\log n)$ edges of F. Since every set F_i contains at most z edges, $|F| = O(z \log n)$. Lastly, since every set $\mathcal{P}'_i$ of paths causes congestion $O(\Delta/\psi)$, the paths in $\mathcal{P}$ cause congestion $O(\Delta \log n/\psi)$. It now remains to bound the running time of the algorithm.

The algorithm performs $O(\log n)$ iterations. Each iteration requires running the Algorithm CutOrCertify from Theorem 8.1.3, which takes time $O\left(n^{1+O(1/r)} \cdot (\log n)^{O(r^2)}\right)$, and Algorithm RouteOrCut-1Pair from Theorem 8.3.8, that takes time $\tilde{O}\left(n\Delta^2/\psi\right)$. Therefore, the total running time of the algorithm is: $\tilde{O}\left(n^{1+O(1/r)} \cdot (\log n)^{O(r^2)} + n\Delta^2/\psi\right)$.

□

8.5.2 Degree Reduction

Assume that we are given a graph $G = (V, E)$ with $|V| = n$ and $|E| = m$, that we view as an input to the BalCutPrune problem. In this subsection we show a deterministic algorithm, that we call ReduceDegree, that has running time $O(m)$, and transforms G into a bounded-degree graph $\hat{G}$. We also provide an algorithm that transforms any sparse balanced cut in a subgraph of $\hat{G}$ into a "nice" cut, that corresponds to a sparse balanced cut in a subgraph of

G.

We first describe Algorithm ReduceDegree for constructing the graph $\hat{G}$. For convenience, we denote $V = \{v_1, \ldots, v_n\}$. For every vertex $v_i \in V$, we let $\deg(v_i)$ denote the degree of v_i in G, and we let $\left\{e_1(v_i), \ldots, e_{\deg(v_i)}(v_i)\right\}$ be the set of edges incident to v, indexed in an arbitrary order. For every vertex $v_i \in V$, we use Lemma 8.2.1 to construct a graph H_i on a set V_i of $\deg(v_i)$ vertices, that is an α_0-expander, for some constant α_0, such that the maximum vertex degree in H_i is at most 9. Recall that the running time of the algorithm for constructing H_i is $O(\deg(v_i))$. We denote the vertices of H_i by $V_i = \left\{u_1(v_i), \ldots, u_{\deg(v_i)}(v_i)\right\}$.

In order to obtain the final graph $\hat{G}$, we start with a disjoint union of all graphs in $\{H_i \mid v_i \in V\}$. All edges lying in such graphs H_i are called *type-1 edges*. Additionally, we add to $\hat{G}$ a collection of type-2 edges, defined as follows. Consider any edge $e = (v, v') \in E$, and assume that $e = e_j(v) = e_{j'}(v')$ (that is, e is the jth edge incident to v and it is the j'th edge incident to v'). We then let $\hat{e}$ be the edge $(u_j(v), u_{j'}(v))$. For every edge $e \in E$, we add the corresponding new edge $\hat{e}$ to graph $\hat{G}$ as a type-2 edge. This concludes the construction of the graph $\hat{G}$, that we denote by $\hat{G} = (\hat{V}, \hat{E})$. Note that the maximum vertex degree in $\hat{G}$ is at most 10, and $|\hat{V}| = 2m$. Moreover, the running time of the algorithm for constructing the graph $\hat{G}$ is $O(m)$.

We say that a subset $S \subseteq \hat{V}$ of vertices is *canonical* iff for every vertex $v_i \in V$, either $V_i \subseteq S$, or $V_i \cap S = \emptyset$. Similarly, we say that a cut (X, Y) in a subgraph of $\hat{G}$ is canonical iff each of X, Y is a canonical subset of $\hat{V}$. The following lemma allows us to convert an arbitrary sparse balanced cuts in a subgraph of $\hat{G}$ into a canonical one.

Lemma 8.5.4

Let $\alpha_0 > 0$ be the constant from Lemma 8.2.1. There is a deterministic algorithm, that we call MakeCanonical, that, given a subgraph $\hat{G}' \subseteq \hat{G}$, where $V(\hat{G}')$ is a canonical vertex set, and a cut (A, B) in $\hat{G}'$, computes, in time $O(m)$, a canonical cut (A', B') in $\hat{G}'$, such that $|A'| \geq |A|/2$, $|B'| \geq |B|/2$, and moreover, if $|E_{\hat{G}}(A, B)| \leq \psi \min\{|A|, |B|\}$, for $\psi \leq \alpha_0/2$, then $|E_{\hat{G}}(A', B')| \leq O(|E_{\hat{G}}(A, B)|)$.

Proof. We start with the cut $(\hat{A}, \hat{B}) = (A, B)$ in graph $\hat{G}'$ and then gradually modify it, by processing the vertices of $V(G)$ one-by-one. When a vertex v_i is processed, if $V_i \cap V(\hat{G}') \neq \emptyset$, we move all vertices of V_i to either $\hat{A}$ or $\hat{B}$. Once every vertex of $V(G)$ is processed, we obtain the final cut (A', B'), that will serve as the output of the algorithm.

Consider an iteration when some vertex $v_i \in V(G)$ is processed, and assume that $V_i \subseteq V(\hat{G}')$. Denote $A_i = A \cap V_i$ and $B_i = B \cap V_i$. If $|A_i| \geq |B_i|$, then we move all vertices of B_i to $\hat{A}$, and otherwise we move all vertices of A_i to $\hat{B}$. Assume w.l.o.g. that the latter happened (the other case is symmetric). Note that the only new edges that are added to the cut $E_{\hat{G}}(\hat{A}, \hat{B})$ are type-2 edges that are incident to the vertices of A_i. The number of such edges is bounded by $|A_i|$. The edges of $E_{H_i}(A_i, B_i)$ belonged to the cut $E_{\hat{G}}(\hat{A}, \hat{B})$ before the current iteration, but they do not belong to the cut at the end of the iteration. Since H_i is an α_0-expander, we get that $|A_i| \leq |E_{H_i}(A_i, B_i)|/\alpha_0$. Therefore, the increase in $|E_{\hat{G}}(\hat{A},\hat{B})|$, due to the current iteration is bounded by $|E_{H_i}(A_i,B_i)|/\alpha_0$. We *charge* the edges of $E_{H_i}(A_i,B_i)$ for this increase; note that these edges will never be charged again. The algorithm terminates once all vertices of $V(G)$ are processed. Let (A',B') denote the final cut $(\hat{A},\hat{B})$. From the above discussion, we are guaranteed that $|E_{\hat{G}}(A',B')| \leq |E_{\hat{G}}(A,B)| + \sum_{v_i\in V(G)} |E_{H_i}(A_i,B_i)|/\alpha_0 \leq O(|E_{\hat{G}}(A,B)|)$.

Next, we claim that $|A'| \geq |A|/2$ and that $|B'| \geq |B|/2$. We prove this for $|A'|$; the proof for $|B'|$ is symmetric. Indeed, assume otherwise. Let $V' \subseteq V$ be the set of all vertices v_i, such that, when the algorithm processed v_i, the vertices of A_i were moved from $\hat{A}$ to $\hat{B}$, and let $n_i = |A_i|$. Then $\sum_{v_i\in V'} n_i > |A|/2$ must hold. Notice however that for a vertex $v_i \in V'$, $|E_{H_i}(A_i,B_i)| \geq \alpha_0|A_i| = \alpha_0 n_i$ must hold. Therefore, graph H_i contributed at least $\alpha_0 n_i$ edges to the original cut $E_{\hat{G}}(A,B)$. Since we are guaranteed that $|E_{\hat{G}}(A,B)| \leq \psi\cdot|A|$, we get that $\sum_{v_i\in V'}\alpha_0 n_i \leq \psi\cdot|A|$, and so $\sum_{v_i\in V'} n_i \leq \psi\cdot|A|/\alpha_0 \leq |A|/2$, since we have assumed that $\psi \leq \alpha_0/2$. But this contradicts the fact that we established before, that $\sum_{v\in V'} n_i > |A|/2$. □

8.5.3 Completing the Proof of Theorem 8.5.1

We prove the following theorem, from which Theorem 8.5.1 immediately follows.

Theorem 8.5.5

There is a universal constant N_0', and a deterministic algorithm, that, given an n-vertex m-edge graph $G=(V,E)$, a parameter $0<\phi<1$, and another parameter $r\geq 1$, such that $m^{1/r}\geq N_0'$, returns a cut (A,B) in G with $|E_G(A,B)|\leq \phi\cdot\mathbf{vol}(G)$, such that:

- either $\mathbf{vol}_G(A),\mathbf{vol}_G(B)\geq \mathbf{vol}(G)/3$; or
- $\mathbf{vol}_G(A)\geq \frac{7}{12}\cdot\mathbf{vol}(G)$, and the graph $G[A]$ has conductance $\phi'\geq \phi/\log^{O(r)} m$.

The running time of the algorithm is $O\left(m^{1+O(1/r)}\cdot(\log m)^{O(r^2)}/\phi^2\right)$.

In order to complete the proof of Theorem 8.5.1, we let c be a large enough constant, so that $m^{1/(c\log m)}\geq N_0'$ holds. We then apply the algorithm from Theorem 8.5.5 to the input graph G and the parameter r. In the remainder of this section we focus on the proof of Theorem 8.5.5.

Proof of Theorem 8.5.5. We denote by $\psi_r(n) = 1/\log^{O(r)} n$ the parameter from Theorem 8.1.3 (that is, when Algorithm CutOrCertify from Theorem 8.1.3 returns a set S of at least $n/2$ vertices, then $\Psi(G[S])\geq\psi_r(n)$ holds). Throughout the proof, we use two parameters: $\psi=\phi/\hat{c}$, and $z=\frac{\phi m}{\hat{c}(\log m)^{\hat{c}r}}$, where $\hat{c}$ is a large constant to be set later. We also set $N_0'=4N_0$, where N_0 is the universal constants from Theorem 8.1.3.

We start by using Algorithm ReduceDegree described in Section 8.5.2, in order to construct, in time $O(m)$, a graph $\hat{G}$ whose maximum vertex degree is bounded by 10, and $|V(\hat{G})|=2m$. Denote $V(G)=\{v_1,\ldots,v_n\}$. Recall that graph $\hat{G}$ is constructed from graph G by replacing each vertex v_i with an α_0-expander H_i on $\deg_G(v_i)$ vertices, where $\alpha_0=\Theta(1)$.

For convenience, we denote the set of vertices of H_i by V_i. Therefore, $V(\hat{G})$ is a union of the sets $V_1, \ldots, V_n$ of vertices. Consider now some subset S of vertices of $\hat{G}$. Recall that we say that S is a *canonical* vertex set iff for every $1 \leq i \leq n$, either $V_i \subseteq S$ or $V_i \cap S = \emptyset$ holds.

The algorithm performs a number of iterations. We maintain a subgraph $\hat{G}' \subseteq \hat{G}$; at the beginning of the algorithm, $\hat{G}' = \hat{G}$. In the ith iteration, we compute a canonical subset $S_i \subseteq V(\hat{G}')$ of vertices, and then update the graph $\hat{G}'$, by deleting the vertices of S_i from it. The iterations are performed as long as $|\bigcup_i S_i| < |V(\hat{G})|/3$.

In order to execute the ith iteration, we consider the current graph $\hat{G}'$, denoting $|V(\hat{G}')| = n'$. Note that, since we assume that $|\bigcup_{i'<i} S_{i'}| < |V(\hat{G})|/3$, we get that $n' \geq 2|V(\hat{G})|/3$. From our choice of parameter N_0', we are guaranteed that $(n')^{1/r} \geq N_0$. We can now apply Lemma 8.5.3 to graph $\hat{G}'$, with the parameters r, ψ and z. Recall that the maximum vertex degree in $\hat{G}'$ is $\Delta \leq 10$. Assume first that the outcome is a cut (X, Y) in $\hat{G}'$ with $|X|, |Y| \geq z/\Delta \geq z/10$ and $\Psi_{\hat{G}'}(X, Y) \leq \psi$. We say that the iteration terminates with a cut in this case. By setting $\hat{c}$ to be a large enough constant, we can ensure that $\psi \leq \alpha_0/2$ where α_0 is the constant from Lemma 8.2.1. We use the algorithm MakeCanonical from Lemma 8.5.4 to compute, in time $O(m)$, a canonical partition (X', Y') of $V(\hat{G}')$, such that $|X'|, |Y'| \geq \Omega(z)$, and $|E_{\hat{G}'}(X', Y')| \leq O(|E_{\hat{G}'}(X, Y)|)$. Assume w.l.o.g. that $|X'| \leq |Y'|$. We are then guaranteed that $|X'| \geq \Omega(z)$, and that for some constant μ, $|E_{\hat{G}'}(X', Y')| \leq \mu\psi|X'|$, or equivalently, $\Psi_{\hat{G}'}(X', Y') \leq \mu\psi$. We set $S_i = X'$, delete the vertices of S_i from $\hat{G}'$, and continue to the next iteration. Observe that set $V(\hat{G}')$ of vertices remains canonical. Otherwise, the outcome of Lemma 8.5.3 is a graph H with $V(H) = V(\hat{G}')$, that is a $\psi_r(n')$-expander, together with a set F of at most $O(z \log n)$ fake edges for $\hat{G}'$, and an embedding of H into $\hat{G}' + F$ with congestion at most $O(\log m/\psi)$, such that every vertex of $\hat{G}'$ is incident to at most $O(\log m)$ edges of F. In this case we say that the iteration terminates with an expander. If an iteration terminates with an expander, then the whole algorithm terminates.

Let i denote the index of the last iteration of the algorithm that terminated with a cut. Recall that one of the following two cases must hold:

- (Case 1): the algorithm had exactly i iterations, every iteration terminated with a cut, and $|\bigcup_{i' \leq i} S_{i'}| \geq |V(\hat{G})|/3$;
- (Case 2): the algorithm had $(i + 1)$ iterations, the first ith iterations terminated with cuts, and the last iteration terminated with an expander.

In either case, let $S = \bigcup_{i'=1}^{i} S_{i'}$. Then S is a canonical vertex set for $\hat{G}$, and moreover, it is easy to verify that:

$$|E_{\hat{G}}(S, \overline{S})| \leq \mu\psi|S| \leq \mu\psi|V(\hat{G})|. \tag{8.1}$$

Assume first that Case 1 happened. Consider the partition (A', B') of $V(\hat{G})$, where $A' = S$ and $B' = V(\hat{G}) \setminus S$. Recall that $|\bigcup_{i'<i} S_{i'}| < |V(\hat{G}')|/3$ held (or we would not have executed the ith iteration). Let $\hat{G}_i$ denote the graph $\hat{G}'$ that served as input to the ith iteration, and let $n_i = |V(\hat{G}_i)|$. Then $n_i \geq 2|V(\hat{G})|/3$. Let (X_i, Y_i) be the cut that was returned by Lemma 8.5.3, and let (X_i', Y_i') be the canonical cut that we obtained in $\hat{G}'$, so that

$S_i = X'_i$. Recall that $|X'_i| \leq |Y'_i|$. It follows that $|Y'_i| \geq |V(\hat{G})|/3$, and $|\bigcup_{i' \leq i} S_{i'}| \geq |V(\hat{G})|/3$. Since $A' = \bigcup_{i' \leq i} S_{i'}$ and $B' = Y'_i$, we get that $|A'|, |B'| \geq |V(\hat{G})|/3$. From Equation (8.1), $|E_{\hat{G}}(A', B')| \leq \psi\mu|V(\hat{G})|$.

Lastly, we obtain a cut (A, B) of $V(G)$ as follows. For every vertex $v_i \in V(G)$, if $V_i \subseteq A'$, then we add v_i to A, and otherwise we add it to B. Since, for every $1 \leq i \leq n$, $|V_i| = \deg_G(v_i)$, it is easy to verify that $\mathbf{vol}(A) = |A'| \geq |V(\hat{G})|/3 = \mathbf{vol}(G)/3$, and similarly $\mathbf{vol}(B) \geq \mathbf{vol}(G)/3$. It is also immediate to verify that $|E_G(A, B)| = |E_{\hat{G}}(A', B')| \leq \mu\psi|V(\hat{G})| = \mu\psi \cdot \mathbf{vol}(G)$. Since $\psi = \phi/\hat{c}$, by letting $\hat{c}$ be a large enough constant, we can ensure that $|E_G(A, B)| \leq \phi \cdot \mathbf{vol}(G)$. We return the cut (A, B) as the outcome of the algorithm.

Assume now that Case 2 happened. Let $\hat{G}_{i+1}$ denote the graph $\hat{G}'$ that served as input to the last iteration. Recall that in this last iteration, the algorithm from Lemma 8.5.3 returned a graph H with $V(H) = V(\hat{G}_{i+1})$, that is a $\psi_r(n')$-expander, where $n' = |V(\hat{G}_{i+1}| \geq 2|V(\hat{G})|/3$, together with a set F of at most $O(z \log n)$ fake edges for $\hat{G}_{i+1}$, and an embedding of H into $\hat{G}_{i+1} + F$ with congestion at most $O(\log m/\psi)$, such that every vertex of $\hat{G}_{i+1}$ is incident to at most $O(\log m)$ edges of F. Let $\hat{G}''$ be the graph obtained from $\hat{G}_{i+1}$, by adding the edges of F to it. Then graph H embeds into $\hat{G}''$ with congestion at most $O(\log m/\psi)$, and so, from Lemma 8.2.5, graph $\hat{G}''$ is a ψ'-expander, for $\psi' = \Omega(\psi_r(n') \cdot \psi/\log m) = \Omega\left(\phi/(\log m)^{O(r)}\right)$.

Recall that all vertex sets $S_1, \ldots, S_i$ are canonical; therefore, the set $V(\hat{G}'')$ of vertices is also canonical. Let G'' be the graph obtained from $\hat{G}''$ as follows. For every vertex $v_j \in V(G)$, if $V_j \subseteq V(\hat{G}'')$, then we contract the vertices of V_j into a single vertex v_j, and remove all self loops. Let $A' = V(G'')$. It is easy to verify that G'' can be obtained from $G[A']$, by adding at most $O(z \log m)$ edges to it – the edges corresponding to the fake edges in F. Moreover, $\mathbf{vol}(A') = |V(\hat{G})| - |S| \geq 2|V(\hat{G})|/3 \geq 2\mathbf{vol}(G)/3$. It is also easy to verify that G'' has conductance at least ψ'. Indeed, consider any cut (X, Y) in G''. This cut naturally defines a cut (X', Y') in $\hat{G}''$: for every vertex $v_i \in A'$, if $v_i \in X$, then we add all vertices of V_i to X', and otherwise we add them to Y'. Then $|X'| = \mathbf{vol}_G(X) \geq \mathbf{vol}_{G''}(X)$, $|Y'| = \mathbf{vol}_G(Y) \geq \mathbf{vol}_{G''}(Y)$, and $|E_{\hat{G}''}(X', Y')| = |E_{G''}(X, Y)|$. Since graph $\hat{G}''$ is a ψ'-expander, we get that $|E_{G''}(X, Y)| \geq |E_{\hat{G}''}(X', Y')| \geq \psi' \min\{|X'|, |Y'|\} \geq \psi' \min\{\mathbf{vol}_{G''}(X), \mathbf{vol}_{G''}(Y)\}$.

In our last step, we get rid of the fake edges in G'' by applying Theorem 8.2.3 to it, with conductance parameter ψ', and the set F of fake edges; (recall that $|F| = O(z \log n)$, and $z = \frac{\phi m}{\hat{c}(\log m)^{\hat{c}r}}$ for some large enough constant $\hat{c}$). In order to be able to use the theorem, we need to verify that $|F| \leq \psi' \cdot |E(G'')|/10$. Since $\psi' = \Omega\left(\phi/(\log m)^{O(r)}\right)$, and $|E(G'')| \geq \Omega(m)$, by letting $\hat{c}$ be a large enough constant, we can ensure that this condition holds. Applying Theorem 8.2.3 to graph G'', with conductance parameter ψ', and the set F of fake edges, we obtain a subgraph $G' \subseteq G'' \setminus F$, of conductance at least $\psi'/6 = \Omega\left(\phi/(\log m)^{O(r)}\right)$. Moreover, if we denote by $A = V(G')$ and $\tilde{B} = V(G'') \setminus V(G')$, then $|E_{G''}(A, \tilde{B})| \leq 4k$ and:

$$\mathbf{vol}_{G''}(\tilde{B}) \leq 8k/\psi' \leq O\left(k \cdot (\log m)^{O(r)}/\phi\right), \tag{8.2}$$

where $k = |F| = O(z \log n)$ is the number of the fake edges. The running time of the algorithm from Theorem 8.2.3 is $\tilde{O}\left(m/\psi'\right) = O\left(m(\log m)^{O(r)}/\phi\right)$. Let $B = V(G) \setminus A$. The algorithm then returns the cut (A, B). We now verify that the cut has all required properties. We have already established that $G[A]$ has conductance at least $\phi/(\log m)^{O(r)}$.

Let $\tilde{S} = B \setminus \tilde{B}$. Then equivalently, we can obtain the set $\tilde{S} \subseteq V(G)$ of vertices from the set $S \subseteq V(\hat{G})$ of vertices (recall that $S = \bigcup_{i'=1}^{i} S_{i'}$) by adding to $\tilde{S}$ every vertex $v_j \in V(G)$ with $V_j \subseteq S$. Since, from Equation (8.1), $|E_{\hat{G}}(S, \overline{S})| \leq \mu\psi|V(\hat{G})|$ for some constant μ, it is easy to verify that:

$$|E_G(\tilde{S}, V(G) \setminus \tilde{S})| \leq \mu\psi \cdot \mathbf{vol}(G) = \mu\phi \cdot \mathbf{vol}(G)/\hat{c}. \tag{8.3}$$

From the above discussion, we are also guaranteed that $|E_{G''}(A, \tilde{B})| \leq 4|F| \leq O(z \log n)$. Since $z = \frac{\phi m}{\hat{c}(\log m)^{\hat{c}r}}$, by letting $\hat{c}$ be a large enough constant, we can ensure that $|E_{G''}(A, \tilde{B})| < \phi m/100 \leq \phi\mathbf{vol}(G)/100$. Therefore, altogether, we get that:

$$|E_G(A, B)| \leq |E_G(A, \tilde{B})| + |E_G(\tilde{S}, V(G) \setminus \tilde{S})| \leq \phi \cdot \mathbf{vol}(G)/100 + \phi\mu \cdot \mathbf{vol}(G)/\hat{c} \leq \phi \cdot \mathbf{vol}(G),$$

if $\hat{c}$ is chosen to be a large enough constant.

Lastly, it remains to verify that $\mathbf{vol}_G(A) \geq \frac{7}{12} \cdot \mathbf{vol}(G)$. Recall that $|\hat{V}(G_{i+1})| \geq 2|V(\hat{G})|/3 \geq 2\mathbf{vol}(G)/3$. Therefore, if we denote by $U = V(G'') = V(G) \setminus \tilde{S}$, then $\mathbf{vol}_G(U) \geq 2\mathbf{vol}(G)/3$. Recall that $A = U \setminus \tilde{B}$, and, from Equation (8.2), $\mathbf{vol}_{G''}(\tilde{B}) \leq O\left(k \cdot (\log m)^{O(r)}/\phi\right) \leq O\left(z \cdot (\log m)^{O(r)}/\phi\right)$. Moreover, $\mathbf{vol}_G(\tilde{B}) \leq \mathbf{vol}_{G''}(\tilde{B}) + E_G(\tilde{S}, \tilde{B}) \leq \mathbf{vol}_{G''}(\tilde{B}) + E_G(U, \tilde{S})$. From Equation (8.3), we get that:

$$\mathbf{vol}_G(\tilde{B}) \leq O\left(z \cdot (\log m)^{O(r)}/\phi\right) + O(\mu\phi\mathbf{vol}(G)/\hat{c}).$$

Since $z = \frac{\phi m}{\hat{c}(\log m)^{\hat{c}r}}$, by letting $\hat{c}$ be a large enough constant, we can ensure that $\mathbf{vol}_G(\tilde{B}) \leq \mathbf{vol}(G)/12$. We then get that $\mathbf{vol}_G(A) \geq |\hat{V}(G_{i+1})| - \mathbf{vol}_G(\tilde{B}) \geq 2\mathbf{vol}(G)/3 - \mathbf{vol}(G)/12 \geq 7\mathbf{vol}(G)/12$.

It now remains to analyze the running time of the algorithm. The time required to construct graph $\hat{G}$ from graph G is $O(m)$. Recall that, if an iteration terminates with a cut, then we delete from $\hat{G}'$ a set of at least $\Omega(z)$ vertices. Therefore, the total number of iterations is bounded by $O(|V(\hat{G})|/z) = O(m/z) = O\left((\log m)^{O(r)}/\phi\right)$. The running time of each iteration is:

$$\tilde{O}\left(m^{1+O(1/r)} \cdot (\log m)^{O(r^2)} + m/\psi\right) = \tilde{O}\left(m^{1+O(1/r)} \cdot (\log m)^{O(r^2)} + m/\phi\right).$$

At the end of each iteration, we employ Lemma 8.5.4 to turn the resulting cut into a canonical one, in time $O(m)$. Therefore, the total running time of the iterations is $\tilde{O}\left(m^{1+O(1/r)} \cdot (\log m)^{O(r^2)}/\phi^2\right)$. Lastly, if Case 2 happens, we employ the algorithm from Theorem 8.2.3, whose running time, as discussed above, is $\tilde{O}\left(m(\log m)^{O(r)}/\phi\right)$. Altogether, the running time of the algorithm is $\tilde{O}\left(m^{1+O(1/r)} \cdot (\log m)^{O(r^2)}/\phi^2\right)$.

8.6 Unweighted Expander Decomposition

Finally, we present the expander decomposition algorithm promised by Theorem 8.6.1 (restated below) using BalCutPrune as a subroutine.

Theorem 8.6.1: Deterministic expander decomposition, unweighted

There is a deterministic algorithm that, given an unweighted graph $G = (V, E)$ and parameters $\epsilon \in (0, 1]$ and $1 \leq r \leq O(\log m)$, computes a (ϵ, ϕ)-expander decomposition of G with $\phi = \Omega(\epsilon/(\log m)^{O(r^2)})$, in time $O\left(m^{1+O(1/r)+o(1)} \cdot (\log m)^{O(r^2)}/\epsilon^2\right)$. Setting $r = (\log n)^{1/3}$, we obtain $\phi = \epsilon/n^{o(1)}$ and time $O(m^{1+o(1)}/\epsilon^2)$.

Proof. We maintain a collection $\mathcal{H}$ of disjoint sub-graphs of G that we call *clusters*, which is partitioned into two subsets, set $\mathcal{H}^A$ of *active clusters*, and set $\mathcal{H}^I$ of *inactive clusters*. We ensure that for each inactive cluster $H \in \mathcal{H}^I$, $\Phi(H) \geq \phi$. We also maintain a set E' of "deleted" edges, that are not contained in any cluster in $\mathcal{H}$. At the beginning of the algorithm, we let $\mathcal{H} = \mathcal{H}^A = \{G\}$, $\mathcal{H}^I = \emptyset$, and $E' = \emptyset$. The algorithm proceeds as long $\mathcal{H}^A \neq \emptyset$, and consists of iterations. For convenience, we denote $\alpha = (\log m)^{r^2}$, and we set $\phi = \epsilon/(c\alpha \cdot \log m)$, for some large enough constant c, so that $\phi = \Omega(\epsilon/(\log m)^{O(r^2)})$ holds.

In every iteration, we apply the algorithm from Corollary 8.5.2 to every graph $H \in \mathcal{H}^A$, with the same parameters α, r, and ϕ. Consider the cut (A, B) in H that the algorithm returns, with $|E_H(A, B)| \leq \alpha\phi \cdot \mathbf{vol}(H) \leq \frac{\epsilon \cdot \mathbf{vol}(H)}{c \log m}$. We add the edges of $E_H(A, B)$ to set E'. If $\mathbf{vol}_H(A), \mathbf{vol}_H(B) \geq \mathbf{vol}(H)/3$, then we replace H with $H[A]$ and $H[B]$ in $\mathcal{H}$ and in $\mathcal{H}^A$. Otherwise, we are guaranteed that $\mathbf{vol}_H(A) \geq \mathbf{vol}(H)/2$, and graph $H[A]$ has conductance at least ϕ. Then we remove H from $\mathcal{H}$ and $\mathcal{H}^A$, add $H[A]$ to $\mathcal{H}$ and $\mathcal{H}^I$, and add $H[B]$ to $\mathcal{H}$ and $\mathcal{H}^A$.

When the algorithm terminates, $\mathcal{H}^A = \emptyset$, and so every graph in $\mathcal{H}$ has conductance at least ϕ. Notice that in every iteration, the maximum volume of a graph in $\mathcal{H}^A$ must decrease by a constant factor. Therefore, the number of iterations is bounded by $O(\log m)$. It is easy to verify that the number of edges added to set E' in every iteration is at most $\frac{\epsilon \cdot \mathbf{vol}(G)}{c \log m}$. Therefore, by letting c be a large enough constant, we can ensure that $|E'| \leq \epsilon \mathbf{vol}(G)$. The output of the algorithm is the partition $\mathcal{P} = \{V(H) \mid H \in \mathcal{H}\}$ of V. From the above discussion, we obtain a valid (ϵ, ϕ)-expander decomposition, for $\phi = \Omega\left(\epsilon/(\log m)^{O(r^2)}\right)$.

It remains to analyze the running time of the algorithm. The running time of a single iteration is bounded by $O\left(m^{1+O(1/r)} \cdot (\log m)^{O(r^2)}/\phi^2\right) = O\left(m^{1+O(1/r)} \cdot (\log m)^{O(r^2)}/\epsilon^2\right)$. Since the total number of iterations is bounded by $O(\log m)$, we get that the total running time of the algorithm is $O\left(m^{1+O(1/r)} \cdot (\log m)^{O(r^2)}/\epsilon^2\right)$. □

8.6.1 Spectral Sparsification

Our deterministic algorithm for computing expander decompositions from Theorem 8.6.1 immediately implies a deterministic algorithm for the original application of expander decompositions: constructing spectral sparsifiers [101]. We require this application in the next section on weighted expander decomposition.

Suppose we are given a undirected weighted n-vertex graph $G = (V, E, \boldsymbol{w})$ (possibly with self-loops). The Laplacian L_G of G is a matrix of size $n \times n$ whose entries are defined as follows:

$$L_G(u,v) = \begin{cases} 0 & u \neq v, (u,v) \notin E \\ -\boldsymbol{w}_{uv} & u \neq v, (u,v) \in E \\ \sum_{\substack{(u,u') \in E: \\ u \neq u'}} \boldsymbol{w}_{uu'} & u = v. \end{cases}$$

We say that a graph H is an *α-approximate spectral sparsifier* for G iff for all $\boldsymbol{x} \in \mathbb{R}^n$, $\frac{1}{\alpha}\boldsymbol{x}^\top L_G \boldsymbol{x} \leq \boldsymbol{x}^\top L_H \boldsymbol{x} \leq \alpha \cdot \boldsymbol{x}^\top L_G \boldsymbol{x}$ holds.

All previous deterministic algorithms for graph sparsification, including those computing cut sparsifiers, exploit explicit potential function-based approach of Batson, Spielman and Srivastava [13]. All previous algorithms that achieve faster running time either perform random sampling [99], or use random projections, in order to estimate the importances of edges [5]. We provide the first deterministic, almost-linear-time algorithm for computing a spectral sparsifier of a *weighted* graph. We emphasize that although all algorithms from previous sections are designed for unweighted graphs, the fact that spectral sparsifiers are "decomposable" allows us to easily reduce the problem on weighted graphs to the one on unweighted graphs.

Corollary 8.6.2: Deterministic spectral sparsifier

There is a deterministic algorithm, that we call `SpectralSparsify` that, given an undirected n-node m-edge graph $G = (V, E, \boldsymbol{w})$ with integral edge weights $\boldsymbol{w}$ bounded by U, and a parameter $1 \leq r \leq O(\log m)$, computes a $(\log m)^{O(r^2)}$-approximate spectral sparsifier H for G, with $|E(H)| \leq O(n \log n \log U)$, in time $O\left(m^{1+O(1/r)} \cdot (\log m)^{O(r^2)} \log U\right)$.

Proof. We first assume that G is unweighted. We compute a $(1/2, \phi)$-expander decomposition $\mathcal{P} = \{V_1, V_2, \ldots, V_k\}$ of G, for $\phi = 1/(\log m)^{O(r^2)}$, using the algorithm from Theorem 8.6.1. Let $\hat{E}$ denote the set of all edges $e \in E(G)$, whose endpoints lie in different sets in the partition $\mathcal{P}$. If $\hat{E} \neq \emptyset$, then we continue the expander decomposition recursively on $G[\hat{E}]$. Notice that the depth of the recursion is bounded by $O(\log m)$. When this process terminates, we obtain a collection $\{G_1, \ldots, G_z\}$ of sub-graphs of G, that are disjoint in their edges, such that $\bigcup_{j=1}^{z} E(G_j) = E(G)$. Moreover, we are guaranteed that for all $1 \leq j \leq z$, graph G_j has conductance are at least $\phi = 1/(\log m)^{O(r^2)}$. It is now enough to compute a spectral sparsifier for each of the resulting graphs $G_1, \ldots, G_z$ separately.

We can now assume that we are given a graph G whose conductance is at least $\phi = 1/(\log m)^{O(r^2)}$, and our goal is to construct a spectral sparsifier for G. In order to do so, we will first approximate G by a "product demand graph" D, that was defined in [64], and then use the construction of [64], that can be viewed as a strengthening of Lemma 8.2.1, in order to sparsify D.

Definition 8.6.3: Product demand graph, Definition G.13 [64]

Given a vector $\boldsymbol{d} \in (\mathbb{R}_{>0})^n$, its corresponding *product demand graph* $H(\boldsymbol{d})$, is a complete weighted graph on n vertices with self-loops, where for every pair i, j of vertices, the weight $\boldsymbol{w}_{ij} = \boldsymbol{d}_i \boldsymbol{d}_j$.

Given an n-node edge-weighted graph $G = (V, E, \boldsymbol{w})$, let $\deg_G \in \mathbb{Z}^n$ be the vector of weighted degrees of every vertex (that includes self-loops), so for all $j \in V$, the jth entry of $\deg_G$ is $\deg_G(j) = \sum_{i \in V} w_{i,j}$. Given an input graph G, we construct a product demand graph $D = \frac{1}{\mathbf{vol}(G)} H(\deg_G)$. It is immediate to verify that the weighted degree vectors of D and G are equal, that is, $\deg_D = \deg_G$.

Next, we need to extend the notion of conductance to weighted graphs with self loops. Consider a weighted graph $H = (V', E', \boldsymbol{w}')$ (that may have self-loops), and let $S \subseteq V'$ be a cut in H. We then let $\delta_H(S) = \sum_{\substack{(u,v) \in E': \\ u \in S, v \notin S}} \boldsymbol{w}'_{u,v}$, and we let $\mathbf{vol}_H(S) = \sum_{v \in S} \sum_{u \in V'} \boldsymbol{w}'_{u,v}$. A weighted conductance of the cut S in H is then: $\frac{\delta_H(S)}{\min\left\{\mathbf{vol}_H(S), \mathbf{vol}_H(\overline{S})\right\}}$, and the conductance of H is the minimum conductance of any cut in H. We need the following observation:

Observation 8.6.4

The weighted conductance of graph D is at least $1/2$.

Proof. Consider any cut S in D. Observe that, from our construction, $\delta_D(S) = \mathbf{vol}_G(S) \cdot \mathbf{vol}_G(\overline{S})/\mathbf{vol}(G)$. It is also easy to see that $\mathbf{vol}_H(S) = \mathbf{vol}_G(S)$. Assume without loss of generality that $\mathbf{vol}_D(S) \leq \mathbf{vol}_D(\overline{S})$, so $\mathbf{vol}_D(\overline{S}) \geq \mathbf{vol}(G)/2$. Then the conductance of the cut S is:

$$\frac{\delta_D(S)}{\mathbf{vol}_D(S)} = \frac{\mathbf{vol}_G(S) \cdot \mathbf{vol}_G(\overline{S})}{\mathbf{vol}(G) \cdot \mathbf{vol}_G(S)} \geq \frac{1}{2}.$$

□

In the following lemma, we show that D is a spectral sparsifier for G.

Lemma 8.6.5

Let D and G be two undirected weighted n-vertex graphs with $V(D) = V(G)$, such that $\deg_D = \deg_G$. Assume further that $\Phi(D), \Phi(G) \geq \phi$ for some conductance threshold ϕ. Then for any real vector $\boldsymbol{x} \in \mathbb{R}^n$: $\frac{\phi^2}{4} \boldsymbol{x}^\top L_G \boldsymbol{x} \leq \boldsymbol{x}^\top L_D \boldsymbol{x} \leq \frac{4}{\phi^2} \boldsymbol{x}^\top L_G \boldsymbol{x}$.

Proof. The normalized Laplacian $\widehat{L}_H$ of a weighted graph H is defined as $W_H^{-1/2} L_H W_H^{-1/2}$, where L_H is the Laplacian of H and W_H is a diagonal weighted-degree matrix, where for every vertex v of H, $(W_H)_{vv} = \deg_H(v)$.

Let $\widehat{L}_D$ and $\widehat{L}_G$ be normalized Laplacians of D and G, respectively. It is well-known that eigenvalues of normalized Laplacians are between 0 and 2. Also, observe that, for any graph H, $L_H \vec{1} = 0$. Therefore, $\widehat{L}_G(\deg_G)^{1/2} = \widehat{L}_D(\deg_G)^{1/2} = 0$. That is, $(\deg_G)^{1/2}$ is in the kernel of both $\widehat{L}_G$ and $\widehat{L}_D$.

Let λ be the second smallest eigenvalue of $\widehat{L}_H$. Then for any vector $\boldsymbol{x}' \perp (\deg_G)^{\frac{1}{2}}$, we have:

$$\frac{\lambda}{2} \boldsymbol{x}'^{\top} \widehat{L}_D \boldsymbol{x}' \leq \lambda \|\boldsymbol{x}'\|^2 \leq \boldsymbol{x}'^{\top} \widehat{L}_G \boldsymbol{x}',$$

since the largest eigenvalue of $\widehat{L}_D$ is at most 2. This implies that, for every vector $\boldsymbol{x} \in \mathbb{R}^n$, $\boldsymbol{x}^{\top} \widehat{L}_G \boldsymbol{x} \geq \frac{\lambda}{2} \boldsymbol{x}^{\top} \widehat{L}_D \boldsymbol{x}$ holds. Indeed, we can write

$$\boldsymbol{x} = \overline{\boldsymbol{x}} + c\,(\deg_G)^{\frac{1}{2}}$$

where $\overline{\boldsymbol{x}} \perp (\deg_G)^{\frac{1}{2}}$ and c is a scalar. This gives:

$$\begin{aligned}
\boldsymbol{x}^{\top} \widehat{L}_G \boldsymbol{x} &= \left(\overline{\boldsymbol{x}} + c\,(\deg_G)^{\frac{1}{2}}\right)^{\top} \widehat{L}_G \left(\overline{\boldsymbol{x}} + c\,(\deg_G)^{\frac{1}{2}}\right) \\
&= \overline{\boldsymbol{x}}^{\top} \widehat{L}_G \overline{\boldsymbol{x}} \\
&\geq \frac{\lambda}{2} \cdot \overline{\boldsymbol{x}}^{\top} \widehat{L}_D \overline{\boldsymbol{x}} \\
&= \frac{\lambda}{2} \cdot \left(\overline{\boldsymbol{x}} + c\,(\deg_G)^{\frac{1}{2}}\right)^{\top} \widehat{L}_D \left(\overline{\boldsymbol{x}} + c\,(\deg_G)^{\frac{1}{2}}\right) \\
&= \frac{\lambda}{2} \cdot \boldsymbol{x}^{\top} \widehat{L}_D \boldsymbol{x}
\end{aligned}$$

where the last equality uses the fact that $\deg_G = \deg_D$. By Cheeger's inequality [6], we have $\lambda \geq \Phi(G)^2/2 \geq \phi^2/2$. Therefore, for any vector $\boldsymbol{x} \in \mathbb{R}^n$:

$$\boldsymbol{x}^{\top} \widehat{L}_G \boldsymbol{x} \geq \frac{\phi^2}{4} \boldsymbol{x}^{\top} \widehat{L}_D \boldsymbol{x} \tag{8.4}$$

We can now conclude that, for any vector $\boldsymbol{x} \in \mathbb{R}^n$:

$$\begin{aligned}
\boldsymbol{x}^\top L_G \boldsymbol{x} &= \boldsymbol{x}^\top W_G^{1/2} \widehat{L}_G W_G^{1/2} \boldsymbol{x} \\
&\geq \frac{\phi^2}{4} \boldsymbol{x}^\top W_G^{1/2} \widehat{L}_D W_G^{1/2} \boldsymbol{x} \\
&= \frac{\phi^2}{4} \boldsymbol{x}^\top W_G^{1/2} W_D^{-1/2} L_D W_D^{-1/2} W_G^{1/2} \boldsymbol{x} \\
&= \frac{\phi^2}{4} \boldsymbol{x}^\top L_D \boldsymbol{x}
\end{aligned}$$

where the first inequality follows by applying Equation (8.4) to vector $x' = W_G^{1/2} x$, and the last equality follows from the fact that $\deg_G = \deg_D$. The proof that $\boldsymbol{x}^\top L_D \boldsymbol{x} \geq \frac{\phi^2}{4} \boldsymbol{x}^\top L_H \boldsymbol{x}$ is similar.

□

Using Lemma 8.6.5 with $\phi = 1/(\log m)^{O(r^2)}$ implies that D is a $\left((\log m)^{O(r^2)}\right)^2 = (\log m)^{O(r^2)}$-approximate spectral sparsifier of H. Finally, a spectral sparsifier for graph D can be constructed in nearly linear time using the following lemma.

Lemma 8.6.6: Lemma G.15, [64]

There exists a deterministic algorithm that, given any demand vector $\boldsymbol{d} \in \mathbb{R}^n$, computes, in time $O(n\epsilon^{-4})$, a graph K with $O(n\epsilon^{-4})$ edges such that $e^{-\epsilon}K$ is an $e^{2\epsilon}$-approximate spectral sparsifier of $H(\boldsymbol{d})$.

By letting $\epsilon = 2$ and $\boldsymbol{d} = \deg_D$ in Lemma 8.6.6, we obtain an 100-approximate spectral sparsifier for graph D (by scaling K), which is in turn a $(\log m)^{O(r^2)}$-approximate spectral sparsifier for graph G. By combining the spectral sparsifiers that we have computed for all sub-graphs of the original input graph G, we obtain an $(\log m)^{O(r^2)}$-approximate spectral sparsifier of the original graph G. The total number of edges in the sparsifier is $O(n \log n)$, as every level of the recursion contributes $O(n)$ edges.

We now analyze the running time of the algorithm. Since the depth of the recursion is $O(\log m)$, running Theorem 8.6.1 takes $O\left(m^{1+O(1/r)} \cdot (\log m)^{O(r^2)}\right)$ time in total. Sparsifying the resulting expanders takes $O(m\text{polylog}(m))$ time. Therefore, the overall running time is bounded by $O\left(m^{1+O(1/r)} \cdot (\log m)^{O(r^2)}\right)$.

For the general (weighted) case, it suffices to decompose the graph by the binary representations of the edge weights and sum the results up: For every edge $e \in E(G)$, let $\boldsymbol{b}_e$ be the binary representation of the weight w_e. For all $1 \leq i \leq \lceil \log(\max_e \boldsymbol{w}_e) \rceil$, we construct an unweighted graph $G^{(i)}$, whose vertex set is V, and edge set contains every edge $e \in E(G)$, such that the ith bit of $\boldsymbol{b}_e$ is 1. Since $\boldsymbol{w}_e \leq U$ for every $e \in E(G)$, there are at most $\lceil \log U \rceil$ such $G^{(i)}$s. By the algorithm for the unweighted case, we compute $(\log m)^{O(r^2)}$-approximate spectral sparsifiers for each $G^{(i)}$. The desired $(\log m)^{O(r^2)}$-approximate spectral sparsifier for

G is $\sum_{i=1}^{\lceil \log(\max_e \boldsymbol{w}_e)\rceil} 2^i G^{(i)}$. This sparsifier contains $\sum_{i=1}^{\lceil \log(\max_e \boldsymbol{w}_e)\rceil} |E(G^{(i)})| = O(n \log n \log U)$ edges. The total running time is $O\left(m^{1+O(1/r)} \cdot (\log m)^{O(r^2)} \log U\right)$. $\square$

8.7 Weighted Expander Decomposition with Custom Demands

In this section, we present our deterministic algorithm for weighted expander decomposition with custom demands on the vertices. In this setting, the input graph is weighted and every vertex $v \in V(G)$ has a non-negative demand $\mathbf{d}(v)$ that is independent of the edge weights. We define the demand of a set $S \subseteq V(G)$ of vertices is $\mathbf{d}(S) = \sum_{v \in S} \mathbf{d}(v)$. Given a subset $S \subseteq V$ of vertices, we denote by $\mathbf{d}_{|S}$ the vector $\mathbf{d}$ of demands restricted to the vertices of S. We start by defining a weighted variant of sparsity and of expander decomposition.

Definition 8.7.1: Weighted sparsity with custom demands

Given a graph $G = (V, E)$ with non-negative weights $w(e) \geq E$ on its edges $e \in E$, and non-negative demands $\mathbf{d}(v) \geq 0$ on its vertices $v \in V$, the $\mathbf{d}$-*sparsity* of a subset $S \subseteq V$ of vertices with $0 < \mathbf{d}(S) < \mathbf{d}(V)$ is:

$$\Psi_G^{\mathbf{d}}(S) = \frac{w(E_G(S, V \setminus S))}{\min\{\mathbf{d}(S), \mathbf{d}(V \setminus S)\}}.$$

The $\mathbf{d}$-*sparsity* of graph G is $\Psi^{\mathbf{d}}(G) = \min_{S \subseteq V : 0 < \mathbf{d}(S) < \mathbf{d}(V)} \Psi_G^{\mathbf{d}}(S)$.

Observe that if $w(e) = 1$ for all $e \in E$ and $\mathbf{d}(v) = \deg(v)$ for all $v \in V$, then this definition is exactly the conductance of the graph. Here, we use the term *sparsity* instead of conductance because traditionally, sparsity concerns the number of vertices in the denominator of the ratio, while conductance uses volume which is closely related to the number of edges. However, for lack of an alternative term, we will stick with the term *expander* to describe a graph of high weighted sparsity. We now define an expander decomposition for the weighted sparsity, which generalizes the standard definition for conductance.

Definition 8.7.2: Weighted expander decomposition with custom demands

Given a graph $G = (V, E, w, \mathbf{d})$ with non-negative weights $w(e) \geq 0$ on its edges $e \in E$, and non-negative demands $\mathbf{d}(v) \geq 0$ on its vertices $v \in V$, a (ϵ, ψ)-*expander decomposition* of G is a partition $\mathcal{P} = \{V_1, \ldots, V_k\}$ of the set V of vertices, such that:

1. For all $1 \leq i \leq k$, the graph $G[V_i]$ has $\mathbf{d}|_{V_i}$-sparsity at least ψ, and
2. $\sum_{i-1}^{k} w(E_G(V_i, V \setminus V_i)) \leq \epsilon \mathbf{d}(V)$.

The main result of this section is an algorithm to compute such a decomposition.

Theorem 8.7.3: Weighted expander decomposition algorithm with custom demands

There is a deterministic algorithm that, given an m-edge graph $G = (V, E)$ with weights $1 \leq w(e) \leq U$ on its edges $e \in E$, and demands $\mathbf{d}(v) \in \{0\} \cup [1, U]$ for its vertices $v \in V$ that are not all zero, together with a parameter $\epsilon \in (0, 1]$ and $r \geq 1$, computes a (ϵ, ψ)-expander decomposition of G, for $\psi = \epsilon / \big(\log^{O(r^4)} m \log U\big)$, in time $m \cdot (mU)^{O(1/r)} \log(mU)$.

8.7.1 Our Techniques

Similarly to the unweighted case, the key subroutine of our expander decomposition algorithm is solving the following $\mathsf{WeightedBalCutPrune}$ problem, a generalization of $\mathsf{BalCutPrune}$ (Definition 8.1.1) that allows both weighted edges and demands on the vertices.

Definition 8.7.4: $\mathsf{WeightedBalCutPrune}$ problem

The input to the α-approximate $\mathsf{WeightedBalCutPrune}$ problem is a graph $G = (V, E)$ with non-negative weights $w(e) \geq 0$ on edges $e \in E$, a nonzero vector $\mathbf{d} \in \mathbb{R}^V_{\geq 0}$ of demands, a sparsity parameter $0 < \psi \leq 1$, and an approximation factor α. The goal is to compute a partition (A, B) of $V(G)$ (where possibly $B = \emptyset$), with $w(E(A, B)) \leq \alpha\psi \cdot \min\{\mathbf{d}(A), \mathbf{d}(B)\}$, such that one of the following hold: either

1. **(Cut)** $\mathbf{d}(A), \mathbf{d}(B) \geq \mathbf{d}(V)/3$; or
2. **(Prune)** $\mathbf{d}(A) \geq \mathbf{d}(V)/2$, and $\Psi^{\mathbf{d}|_A}(G[A]) \geq \psi$.

We remark that the guarantee $w(E(A, B)) \leq \alpha\psi \cdot \min\{\mathbf{d}(A), \mathbf{d}(B)\}$ is stronger than what we would obtain if we directly translated $\mathsf{BalCutPrune}$. The latter only requires that $|E(A, B)| \leq \alpha\psi \cdot \mathbf{vol}(G)$ in their setting, which would translate to $w(E(A, B)) \leq \alpha\psi \cdot \mathbf{d}(V)$ in our setting.

Theorem 8.7.5: Deterministic $\mathsf{WeightedBalCutPrune}$ algorithm

There is a deterministic algorithm that, given an m-edge connected graph $G = (V, E)$ with edge weights $1 \leq w(e) \leq U$ for all $e \in E$ and demands $\mathbf{d}(v) \in \{0\} \cup [1, U]$ for all $v \in V$ that are not all zero, together with parameters $0 < \psi \leq 1$ and $r \geq 1$, solves the $(\log^{O(r^4)} m)$-approximate $\mathsf{WeightedBalCutPrune}$ problem in time $m \cdot (mU)^{O(1/r)}$.

Unlike in the unweighted case, we still do not solve $\mathsf{WeightedBalCutPrune}$ directly. Rather, we reduce it to another problem, the Most-Balanced Cut problem, and then provide a bi-criteria approximation algorithm for Most-Balanced Cut, this time based on recursively applying the j-tree framework of Madry [77]. In Section 8.7.3, we then show our algorithm for Weighted Most-Balanced Cut can be used in order to approximately solve the

WeightedBalCutPrune problem.

Definition 8.7.6: (s, b)-most-balanced ψ-sparse cut

Given a graph $G = (V, E)$ and parameters $s, b \geq 1$, a set $S \subseteq V$ with $\mathbf{d}(S) \leq \mathbf{d}(V)/2$ is a *(s, b)-most-balanced ψ-sparse cut* if it satisfies:

1. $w(S, V \setminus S) \leq \psi \cdot \mathbf{d}(S)$.
2. Define $\psi^* := \psi/s$ and let $S^* \subseteq V$ be the set with maximum $\mathbf{d}(S^*)$ out of all sets S' satisfying $w(S', V \setminus S') \leq \psi^* \cdot \min\{\mathbf{d}(S'), \mathbf{d}(V \setminus S')\}$ and $\mathbf{d}(S') \leq \mathbf{d}(V)/2$. Then, $\mathbf{d}(S) \geq \mathbf{d}(S^*)/b$.

Let us first motivate why we consider a completely different recursive framework based on recursive j-trees [77] instead of the recursive KKOV cut-matching game framework [58] as used in the unweighted case. This is because KKOV recursion scheme does not generalize easily to the weighted setting. The main issue that in a weighted graph, the flows constructed by the matching player cannot be decomposed into a small number of paths; the only bound we can prove is at most m paths by standard flow decomposition arguments. Hence, the graphs constructed by the cut player are not any sparser, preventing us from obtaining an efficient recursive bound. Madry's j-tree framework, on the other hand, generalizes smoothly to weighted instances and can even be adapted to solve the sparsest cut problem with general demands, for which Madry provided efficient randomized algorithms in his original paper [77].

Below, we give a high-level description of Madry's approach. But first, let us state the definition of j-trees as follows.

Definition 8.7.7: j-tree

A graph G is a j-tree if it is a union of:

- a subgraph H of G (called the *core*), induced by a set V_H of at most j vertices; and
- a forest (that we refer to as *peripheral forest*), where each connected component of the forest contains exactly one vertex of V_H. For each core vertex $v \in V_H$, we let $T_G(v)$ denote the unique tree in the peripheral forest that contains v. When the j-tree G is unambiguous, we may use $T(v)$ instead.

In Madry's approach, the input graph is first decomposed into a small number of j-trees (formally stated in Lemma 8.2.7), so that it suffices to solve the problem on each j-tree and take the best solution. For a given j-tree, one key property of the generalized sparsest cut problem is that either the optimal solution only cuts edges of the core, or it only cuts edges of the peripheral forest. Therefore, the algorithm can solve two separate problems, one on the core and one on the peripheral forest. The former becomes a recursive call on a graph of j vertices, and the latter simply reduces to solving the problem on a tree.

This same strategy almost directly translates over to the Weighted Most-Balanced Cut problem. The main additional difficulty is in ensuring the additional *balanced* guarantee in our Weighted Most-Balanced Cut problem, which is the biggest technical component of this section. We remark that our algorithm for computing the weighted most-balanced sparse cut is a modification of the algorithm in Section 8 of [41]. In particular, the algorithms `WeightedBalCut` and `RootedTreeBalCut` presented below are direct modifications of Algorithm 4 and Algorithm 5 in Section 8 of [41], respectively. Still, we assume no familiarity with that paper and make no references to it.

8.7.2 The WeightedBalCutPruneAlgorithm

Our algorithm `WeightedBalCut` first invokes Lemma 8.2.7 to approximately decompose the input graph G into t many j-trees, where $j = O(m/t)$ and t is small (say, m^ϵ for some constant $\epsilon > 0$). Since the distribution of j-trees approximates G, it suffices to solve the Weighted Most-Balanced Cut problem on each j-tree separately and take the best overall. For a given j-tree H, the algorithm computes two types of cuts—one that only cuts edges in the core of H, and one that only cuts edges of the peripheral forest of H—and takes the one with better weighted sparsity. In our analysis (specifically Lemma 8.7.9), we prove our correctness by showing that for any cut S of the j-tree $H = (V_H, E_H)$, there exists a cut S' that

1. either only cuts core edges or only cuts peripheral edges, and
2. has weighted sparsity and balance comparable to those of H, i.e., $w_H(E_H(S', V_H \setminus S')) \leq O(w_H(E_H(S, V_H \setminus S)))$ and $\mathbf{d}(S') \geq \Omega(\mathbf{d}(S))$.

To compute the best way to cut the core, the algorithm first contracts all edges in the peripheral forest, summing up the demands on the contracted vertices. This leaves a graph of $j = O(m/t)$ vertices, but the number of edges can still be $\Omega(m)$. To ensure the number of edges also drops by a large enough factor, the algorithm sparsifies the core using Corollary 8.6.2, computing a sparse graph with only $\tilde{O}(m/t)$ edges that α-approximates all cuts of the core for some $\alpha = (\log m)^{O(r^2)}$. Finally, the algorithm recursively solves the problem on the sparsified core. The approximation factor blows up by polylog(m) per recursion level, but the number of edges decreases by roughly $t = m^\epsilon$, so over the $O(1/\epsilon)$ recursion levels, the overall approximation factor becomes $(\log m)^{O(r^2/\epsilon)}$, which is $n^{o(1)}$ appropriate choices of r and ϵ.

The algorithm for cutting the peripheral forest is much simpler and non-recursive. The algorithm first contracts the core of H, obtaining a tree in which to compute an approximate Weighted Most-Balanced Cut. Then, `RootedTreeBalCut` roots the tree at an appropriately chosen "centroid" vertex and greedily adds subtrees of small enough sparsity into a set S until either $\mathbf{d}(S)$ is large enough, or no more sparse cuts exist.

```
WeightedBalCut$(G, \psi, \psi^*, b)$ with $\psi \geq \psi^*$ and $b \geq 1$, and $G$ has demands $\mathbf{d}$:
  1. Fix an integer $r \geq 1$ and parameter $t = \left\lceil m_0^{1/r} (\log m)^{O(1)} \log^2 U \right\rceil$, where $m_0$ is the
     number of edges in the *original* input graph to the recursive algorithm, $m \leq m_0$ is
     the number of edges of the input graph $G$ to the current recursive call, and $U$ is the
     capacity ratio of $G$.
  2. Fix parameters $\alpha = (\log m)^{O(r^2)}$ as the approximation factor from Corollary 8.6.2, and
     $\beta = O(\log m (\log \log m)^{O(1)})$ as the congestion factor from Lemma 8.2.7.
  3. Compute $O(m/t)$-trees $G_1, \ldots, G_t$ using Lemma 8.2.7 with $G$ and $t$ as input. For each
     $i$, let $K_i$ denote the vertex set in the core of $G_i$
  4. For each $i \in [t]$:
     (a) $H_i \leftarrow G_i[K_i]$ with demands $\mathbf{d}_{H_i}$ on $K_i$ as $\mathbf{d}_{H_i}(v) = \sum_{u \in V(T_{G_i}(v))} \mathbf{d}(u)$ (so that
         $\mathbf{d}_{H_i}(K_i) = \mathbf{d}(V)$).
     (b) $H_i' \leftarrow \alpha$-approximate spectral sparsifier of $H_i$ (with the same demands)
     (c) $S_{H_i}' \leftarrow$ WeightedBalCut$(H_i', \psi/\alpha, 3\alpha\beta\psi^*, b/3)$
     (d) $S_{H_i} \leftarrow S_{H_i}'$ with each vertex $v$ replaced with $V(T_{G_i}(v))$ (see Definition 8.7.7)
     (e) Construct a tree $T_i = (V_{T_i}, E_{T_i}, w_{T_i})$ with demands $\mathbf{d}_{T_i}$ as follows: Starting with
         $G_i$, contract $K_i$ into a single vertex $k_i$ with demand $\mathbf{d}(K_i)$. All other vertices have
         demand $\mathbf{d}(v)$ (so that $\mathbf{d}_{T_i}(V_{T_i}) = \mathbf{d}(V)$).
     (f) Root $T_i$ at a vertex $r_i \in V_{T_i}$ such that every subtree rooted at a child of $r_i$ has
         total weight at most $\mathbf{d}_{T_i}(V)/2 = \mathbf{d}(V)/2$.
     (g) $S_{T_i}' \leftarrow$ RootedTreeBalCut$(T_i, r_i, \psi)$
     (h) $S_{T_i} \leftarrow S_{T_i}'$ with the vertex $k_i$ replaced with $K_i$ if $k_i \in S_{T_i}'$
  5. Of all the cuts $S = S_{H_i}$ or $S = S_{T_i}$ computed satisfying $w(S, V \setminus S) \leq \psi \cdot \min\{\mathbf{d}(S), \mathbf{d}(V \setminus S)\}$, consider the set $S$ with maximum $\min\{\mathbf{d}(S), \mathbf{d}(V \setminus S)\}$, and output $S$ if $\mathbf{d}(S) \leq$
     $\mathbf{d}(V \setminus S)$ and $V \setminus S$ otherwise. If no cut $S$ satisfies $w(S, V \setminus S) \leq \psi \cdot \min\{\mathbf{d}(S), \mathbf{d}(V \setminus S)\}$,
     then return $\emptyset$.
```

```
RootedTreeBalCut($T = (V_T, E_T, w_T), r, \psi_T$):
    0. Assumption: $T$ is a weighted tree with demands $\mathbf{d}_T$. The tree is rooted at a root $r$
       such that every subtree $V_u$ rooted at a vertex $u \in V_T \setminus \{r\}$ has total demand $\mathbf{d}_T(V_u) \leq \mathbf{d}_T(V_T)/2$.
       Output: a set $S \subseteq V_T$ satisfying the conditions of Lemma 8.7.10.
    1. Find all vertices $u \in V_T \setminus \{r\}$ such that if $V_u$ is the vertices in the subtree rooted at $u$,
       then $w_T(E[V_u, V_T \setminus V_u])/\mathbf{d}_T(V_u) \leq 2\psi_T$. Let this set be $X$.
    2. Let $X^{\uparrow}$ denote all vertices $u \in X$ without an ancestor in $X$ (that is, there is no
       $v \in X \setminus \{u\}$ with $u \in T_v$).
    3. Starting with $S = \emptyset$, iteratively add the vertices $V_u$ for $u \in X^{\uparrow}$. If $\mathbf{d}_T(S) \geq \mathbf{d}_T(V_T)/4$
       at any point, then terminate immediately and output $S$. Otherwise, output $S$ at the
       end.
```

We now analyze our algorithm `WeightedBalCut` by showing the following:

Lemma 8.7.8

Fix parameters $b \geq 6$, $\psi^* > 0$, and $\psi \geq 12\beta \cdot \psi^*$ for β as defined in Line Item 2 of `WeightedBalCut` algorithm. `WeightedBalCut` outputs a $(\psi/\psi^*, b)$-most-balanced $(\psi, \mathbf{d})$-sparse cut.

We now state our structural statement on cuts in j-trees: for each j-tree G_i, either the core H_i contains a good balanced cut or the "peripheral" tree T_i (produced by contracting the core) does.

Lemma 8.7.9

Fix $i \in [t]$, and let $S^* \subseteq V$ be any cut with $\mathbf{d}(S^*) \leq \mathbf{d}(V)/2$. For simplicity, define $K = K_i$, $k = k_i$, $T = T_i$, $r = r_i$, and $H = H_i$. One of the following must hold:

1. There exists a cut $S^*_T \subseteq V_T$ in T satisfying $w_T(E_T(S^*_T, V_T \setminus S^*_T)) \leq w(E_{G_i}(S^*, V \setminus S^*))$ and $\mathbf{d}(S^*)/2 \leq \mathbf{d}_T(S^*_T) \leq 2\mathbf{d}(V)/3$, and S^*_T is the disjoint union of subtrees of T rooted at r.
2. There exists a cut $S^*_H \subseteq K$ in core H satisfying $w_H(E_H(S^*_H, K \setminus S^*_H)) \leq w(E_{G_i}(S^*, V \setminus S^*))$ and $\min\{\mathbf{d}_H(S^*_H), \mathbf{d}_H(K \setminus S^*_H)\} \geq \mathbf{d}(S^*)/3$.

The statement itself should not be surprising. If S^* only cuts edges in the peripheral forest of G_i, then the cut survives when we contract the core H to form the tree T, and its $\mathbf{d}_T$-sparsity is the same as its original $\mathbf{d}$-sparsity. Likewise, if S^* only cuts edges in the core H, then the cut survives when we contract the all edges in the peripheral forest to form K, and its $\mathbf{d}_H$-sparsity is the same as its original $\mathbf{d}$-sparsity. The difficulty is handling the possibility that S^* cuts both peripheral forest edges and core edges, which we resolve

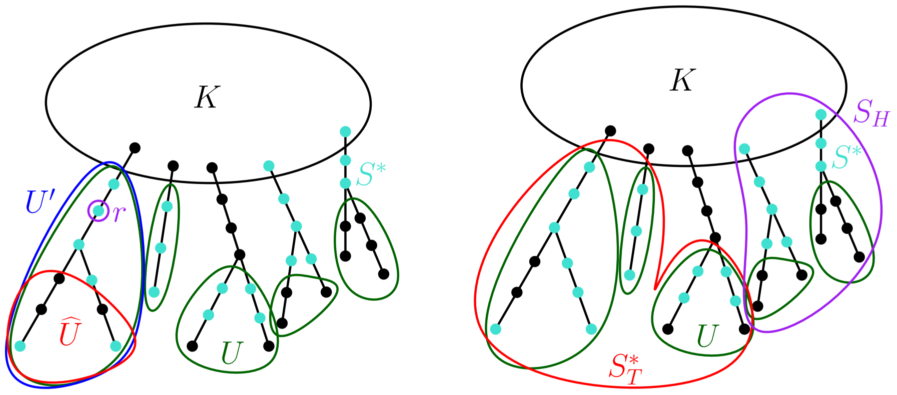

Figure 8.1: Left: Cases 1a and 1b of Lemma 8.7.9. The set S^* is the cyan vertices. Right: Cases 2a and 2b.

through some casework below.

Proof. We need a new notation. For a j-tree G_i and a vertex v on peripheral forest F, we define $c_{G_i}(v)$ as the unique vertex shared by F and the core H of G_i.

Let $S^* \subseteq V$ the set as described in Definition 8.7.6 ($w(S^*, V \setminus S^*) \leq \psi^* \cdot \min\{\mathbf{d}(S^*), \mathbf{d}(V \setminus S^*)\}$). Let U be the vertices $u \in V$ whose (unique) path to $c_{G_i}(u)$ in F contains at least one edge in $E_{G_i}(S^*, V \setminus S^*)$. In Figure 8.1, U is the set of vertices with green circle around. Note that $U \cap K = \emptyset$ and $E_{G_i}(U, V \setminus U) \subseteq E_{G_i}(S^*, V \setminus S^*)$. Observe further that U is a union of subtrees of T rooted at k (not r). This is because, when we root the tree T at k, for each vertex $u \in U$, its entire subtree is contained in U.

Case 1: $r \in U$. In this case, we will construct a cut in the tree T to fulfill condition (1). Let F be the peripheral forest of G_i (see Definition 8.7.7) and let T' be the tree in F that contains r. Define $U' = T' \cap U$ (Figure 8.1 left). In words, U' contains all vertices of U in the tree of F that contains r. Let us re-root T at vertex r, so that the vertices in $V_T \setminus U'$ now form a subtree. We now consider a few sub-cases based on the size of U'.

Case 1a: $r \in U$ and $\mathbf{d}_T(U') \leq 3\mathbf{d}(V)/4$. Define $S_T^* \subseteq V_T$ as $S_T^* := V_T \setminus U'$. By our selection of r,

$$\mathbf{d}_T(S_T^*) = \mathbf{d}_T(V_T \setminus U') \leq \frac{\mathbf{d}(V)}{2}.$$

Moreover,

$$\mathbf{d}_T(S_T^*) = \mathbf{d}_T(V_T \setminus U') \geq \frac{\mathbf{d}(V)}{4} \geq \frac{\mathbf{d}(S^*)}{2}$$

and

$$E_T(S_T^*, V \setminus S_T^*) \subseteq E_{G_i}(U, V \setminus U) \subseteq E_{G_i}(S^*, V \setminus S^*),$$

fulfilling condition (1).

Case 1b: $r \in U$ and $\mathbf{d}_T(U') \geq 3\mathbf{d}(V)/4$. Define $\widehat{U}$ as all vertices $u \in U'$ whose (unique) tree path to root (r) contains at least one vertex not in S^* (possibly u itself). As this set contains all vertices in U' not in S^*, we have $\widehat{U} \supseteq U' \setminus S^*$, and in turn

$$\begin{aligned}\mathbf{d}_T\left(\widehat{U}\right) \geq \mathbf{d}_T\left(U' \setminus S^*\right) = \mathbf{d}_T\left(U'\right) - \mathbf{d}_T\left(U' \cap S^*\right) \geq \mathbf{d}_T\left(U'\right) - \mathbf{d}\left(S^*\right) \\ \geq \frac{3\mathbf{d}\left(V\right)}{4} - \frac{\mathbf{d}\left(V\right)}{2} = \frac{\mathbf{d}(V)}{4}.\end{aligned}$$

Moreover, $\widehat{U}$ is a union of subtrees of T rooted at r and satisfies

$$E_{G_i}\left(\widehat{U}, V \setminus \widehat{U}\right) \subseteq E_{G_i}\left(S^*, V \setminus S^*\right).$$

By our choice of r, each subtree T' of U satisfies $\mathbf{d}_T(V(T')) \leq \mathbf{d}(V)/2$. We perform one further case work based on the largest size of one of these subtrees to show that we can find a tree cut that satisfies condition (1).

- If there exists a subtree $T' \subseteq \widehat{U}$ with $\mathbf{d}_T(V(T')) \geq \mathbf{d}(V)/4$, then set $S_T^* := V(T')$.
- Otherwise, since $\mathbf{d}_T(\widehat{U}) \geq \mathbf{d}(V)/4$, we can greedily select a subset of subtrees of $\widehat{U}$ with total $\mathbf{d}(\cdot)$ value in the range $[\mathbf{d}(V)/4, \mathbf{d}(V)/2]$, and set S_T^* as those vertices.

In both cases we have

$$\frac{\mathbf{d}\left(S^*\right)}{2} \leq \frac{\mathbf{d}(V)}{4} \leq \mathbf{d}_T\left(S_T^*\right) \leq \frac{\mathbf{d}\left(V\right)}{2}$$

which gives the volume condition on S^*, and the cut size bound follows from $w(E_T(S_T^*, V \setminus S_T^*)) \leq w(E_{G_i}(\widehat{U}, V \setminus \widehat{U}))$.

Case 2: $r \notin U$. In this case, we will cut either the tree T or the core H depending on a few further sub-cases.

Case 2a: $r \notin U$ and $\mathbf{d}_T(U) \geq \mathbf{d}(V)/6$. Since $r \notin U$, every subtree in U has weight at most $\mathbf{d}(V)/2$. Let U' be a subset of these subtrees of total $\mathbf{d}(\cdot)$ value in the range $[\mathbf{d}(V)/6, 2\mathbf{d}(V)/3]$. Define the tree cut $S_T^* := U'$, which satisfies

$$\frac{\mathbf{d}(V)}{2} \geq \mathbf{d}_T(S_T^*) \geq \frac{\mathbf{d}(V)}{6} \geq \frac{\mathbf{d}(S^*)}{3}$$

and

$$E_T(S_T^*, V \setminus S_T^*) \subseteq E_{G_i}(U, V \setminus U) \subseteq E_{G_i}(S^*, V \setminus S^*),$$

fulfilling condition (1).

Case 2b: $r \notin U$ and $\mathbf{d}_T(U) < \mathbf{d}(V)/6$. In this case, let $S := S^* \cup U$, which satisfies

$$\mathbf{d}(S^*) \le \mathbf{d}(S) \le \mathbf{d}(S^*) + \mathbf{d}_T(U) \le \mathbf{d}(S^*) + \mathbf{d}(V)/6 \le 2\mathbf{d}(V)/3$$

and $E_{G_i}(S, V \setminus S) \subseteq E_{G_i}(S^*, V \setminus S^*)$. Next, partition S into S_H and S_T^* according to Figure 8.1, where S_H consists of the vertices of all connected components of $G_i[S]$ that intersect K, and $S_T^* := S \setminus S_H$ is the rest. We have

$$E_{G_i}(S_H, V \setminus S_H) \subseteq E_{G_i}(S^*, V \setminus S^*) \qquad \text{and} \qquad E_{G_i}(S_T^*, V \setminus S_T^*) \subseteq E_{G_i}(S^*, V \setminus S^*).$$

Observe that S_T^* is a tree cut, and S_H is a core cut since it does not cut any edges of the peripheral forest. We will select either S_T^* or S_H based on one further case work.

Since $\mathbf{d}(S_T^*) + \mathbf{d}(S_H) = \mathbf{d}(S)$, we can case on whether $\mathbf{d}_T(S_T^*) \ge \mathbf{d}(S)/2$ or $\mathbf{d}(S_H) \ge \mathbf{d}(S)/2$.

- If $\mathbf{d}_T(S_T^*) \ge \mathbf{d}(S)/2$, then the set S_T^* satisfies condition (1).
- Otherwise, $\mathbf{d}(S_H) \ge \mathbf{d}(S)/2$. Since $E_{G_i}(S_H, V \setminus S_H)$ does not contain any edges in the peripheral forest F, we can "contract" the peripheral forest to obtain the set $S_H^* := \{c_{G_i}(v) : v \in S_H\} \subseteq K$ such that S_H is the vertices in the trees in F intersecting S_H^*. This also means that $V \setminus S_H$ is the vertices in the trees of F intersecting $K \setminus S_H^*$. It remains to show that S_H^* fulfills condition (2). We have

  $$w_H(E_H(S_H^*, K \setminus S_H^*)) = w(E_{G_i}(S_H, V \setminus S_H)) \le w(E_{G_i}(S^*, V \setminus S^*))$$

  and
  $$\min\{\mathbf{d}_H(S_H^*), \mathbf{d}_H(K \setminus S_H^*)\} = \min\{\mathbf{d}(S_H), \mathbf{d}(V \setminus S_H)\}.$$

  It remains to show that $\min\{\mathbf{d}(S_H), \mathbf{d}(V \setminus S_H)\} \ge \mathbf{d}(S^*)/3$. This is true because $\mathbf{d}(S_H) \ge \mathbf{d}(S)/2 \ge \mathbf{d}(S^*)/2$ and $\mathbf{d}(S_H) \le \mathbf{d}(S) \le 2\mathbf{d}(V)/3$ which means that $\mathbf{d}(V \setminus S_H) \ge \mathbf{d}(V)/3 \ge \mathbf{d}(S^*)/3$.

□

If the graph H contains a good balanced cut, then intuitively, the demands $\mathbf{d}_H$ are set up so that the recursive call on H' will find a good cut as well. The lemma below shows that if the tree T contains a good balanced cut, then `RootedTreeBalCut` will perform similarly well.

Lemma 8.7.10

`RootedTreeBalCut`$(T = (V_T, E_T, w_T), \mathbf{d}_T, r, \psi_T)$ can be implemented to run in $O(|V_T|)$ time. The set S output satisfies $\psi_T^{\mathbf{d}_T}(S) = w_T(E_T(S, V_T \setminus S))/\min\{\mathbf{d}_T(S), \mathbf{d}_T(V_T \setminus S)\} \leq 6\psi_T$. Moreover, for any set S^* with $w_T(E_T(S^*, V_T \setminus S^*))/\mathbf{d}_T(S^*) \leq \psi_T$ and $\mathbf{d}_T(S^*) \leq 2\mathbf{d}_T(V_T)/3$, and which is composed of vertex-disjoint subtrees rooted at vertices in T, we have $\min\{\mathbf{d}_T(S), \mathbf{d}_T(V_T \setminus S)\} \geq \mathbf{d}_T(S^*)/3$.

Proof. Clearly, every line in the algorithm can be implemented in linear time, so the running time follows. We focus on the other properties.

Every set of vertices V_u added to S satisfies $w_T(E_T(V_u, V_T \setminus V_u))/\mathbf{d}_T(V_u) \leq 2\psi_T$. Also, the added sets V_u are vertex-disjoint, so $w_T(E_T(S, V_T \setminus S)) = \sum_{V_u \subseteq S} w_T(E_T(V_u, V_T \setminus V_u))$. This means that `RootedTreeBalCut` outputs S satisfying $w_T(E_T(S, V_T \setminus S))/\mathbf{d}_T(S) \leq 2\psi_T$. Since every set V_u has total weight at most $\mathbf{d}_T(V_T)/2$, and since the algorithm terminates early if $\mathbf{d}_T(S) \geq \mathbf{d}_T(V_T)/4$, we have $\mathbf{d}_T(S) \leq 3\mathbf{d}_T(V_T)/4$. This means that $\min\{\mathbf{d}_T(S), \mathbf{d}_T(V_T \setminus S)\} \geq \mathbf{d}_T(S)/3$, so $w_T(E_T(S, V_T \setminus S))/\min\{\mathbf{d}_T(S), \mathbf{d}_T(V_T \setminus S)\} \leq 3w_T(E_T(S, V_T \setminus S))/\mathbf{d}_T(S) \leq 6\psi_T$.

It remains to prove that S is balanced compared to S^*. There are two cases. First, suppose that the algorithm terminates early. Then, as argued above, $\min\{\mathbf{d}_T(S), \mathbf{d}_T(V_T \setminus S)\} \geq \mathbf{d}_T(V_T)/4$, which is at least $(2\mathbf{d}_T(V_T)/3)/3 \geq \mathbf{d}_T(S^*)/3$, so $\min\{\mathbf{d}_T(S), \mathbf{d}_T(V_T \setminus S)\} \geq \mathbf{d}_T(S^*)/3$.

Next, suppose that S does not terminate early. From the assumption of S^*, there are sets $S_1^*, \ldots, S_\ell^*$ of vertices in the (vertex-disjoint) subtrees that together compose S^*, that is, $\bigcup_i S_i^* = S^*$. Note that $E_T(S_i^*, V_T \setminus S_i^*)$ is a single edge in E_T for each i. Suppose we reorder the sets S_i^* so that $S_1^*, \ldots, S_q^*$ are the sets that satisfy $w_T(E_T(S_i^*, V_T \setminus S_i^*))/\mathbf{d}_T(S_i^*) \leq 2\psi_T$. From the assumption on S^*, we have $w_T(E_T(S^*, V_T \setminus S^*))/\mathbf{d}_T(S^*) \leq \psi_T$, by a Markov's inequality-like argument, we must have $\sum_{i \in [q]} \mathbf{d}_T(S_i^*) \geq (1/2) \sum_{i \in [\ell]} \mathbf{d}_T(S_i^*) = \mathbf{d}_T(S^*)/2$. Observe that by construction of $X^\uparrow$, each of the subsets $S_1^*, \ldots, S_q^*$ is inside V_u for some $u \in X^\uparrow$. Therefore, the set S that `RootedTreeBalCut` outputs satisfies $\mathbf{d}_T(S) \geq \sum_{i \in [q]} \mathbf{d}_T(S_i^*) \geq \mathbf{d}_T(S^*)/2$. $\square$

Finally, we prove Lemma 8.7.8:

Proof (Lemma 8.7.8). Let $S^* \subseteq V$ be the set for G as described in Definition 8.7.6 with parameters $s = \psi/\psi^*$ and b; that is, it is the set with maximum $\mathbf{d}(S^*)$ out of all sets S' satisfying $\Psi_G^{\mathbf{d}}(S') \leq \psi^*$ and $\mathbf{d}(S') \leq \mathbf{d}(V)/2$. If $\mathbf{d}(S^*) = 0$, then the output of `WeightedBalCut` always satisfies the definition of (s, b)-most-balanced ψ-sparse cut, even if it outputs $\emptyset$. So for the rest of the proof, assume that $\mathbf{d}(S^*) > 0$, so that $\Psi_G^{\mathbf{d}}(S^*)$ and $\Psi_{G_i}^{\mathbf{d}}(S^*)$ are well-defined.

By Lemma 8.2.7, there exists $i \in [t]$ such that $w(E_{G_i}(S^*, V \setminus S^*)) \leq \beta \cdot w(E_G(S^*, V \setminus S^*))$, which means that

$$\Psi_{G_i}^{\mathbf{d}}(S^*) \leq \beta \cdot \Psi_G^{\mathbf{d}}(S^*) \leq \beta \cdot \psi^*.$$

For the rest of the proof, we focus on this i, and define $K = K_i$, $H = H_i$, and $T = T_i$. We break into two cases, depending on which condition of Lemma 8.7.9 is true:

1. Suppose condition (1) is true for the cut S^*_T. Then, since $w_T(E_T(S^*_T, V_T \setminus S^*_T)) \leq w(E_{G_i}(S^*, V \setminus S^*))$ and $\mathbf{d}_T(S^*_T) \geq \mathbf{d}(S^*)/2$, we have
$$\frac{w_T(E_T(S^*_T, V_T \setminus S^*_T))}{\mathbf{d}_T(S^*_T)} \leq \frac{w(E_{G_i}(S^*, V \setminus S^*))}{\mathbf{d}(S^*)/2} \leq 2\Psi^{\mathbf{d}}_{G_i}(S^*) \leq 2\beta \cdot \psi^*.$$
Also, $\mathbf{d}_T(S^*_T) \leq 2\mathbf{d}(V)/3 = 2\mathbf{d}_T(V_T)/3$. Let S'_T be the cut in T that `RootedTreeBalCut` outputs and let S_T the corresponding cut in G_i after the uncontraction in Step Item 4h. Applying Lemma 8.7.10 with $\psi_T = 2\beta \cdot \psi^*$, the cut S'_T satisfies $\Psi^{\mathbf{d}_T}_T(S'_T) \leq 6\psi_T = 12\beta \cdot \psi^*$ and $\min\{\mathbf{d}_T(S'_T), \mathbf{d}_T(V_T \setminus S'_T)\} \geq \mathbf{d}_T(S^*_T)/3$. By construction, $\mathbf{d}(S_T) = \mathbf{d}(S'_T) \geq \mathbf{d}_T(S^*_T)/3 \geq \mathbf{d}(S^*)/6 \geq \mathbf{d}(S^*)/b$ and $\Psi^{\mathbf{d}}_{G_i}(S_T) = \Psi^{\mathbf{d}_T}_T(S'_T) \leq 12\beta \cdot \psi^* \leq \psi$.
2. Suppose condition (2) is true for the cut S^*_H. Since $w_H(E_H(S^*_H, K \setminus S^*_H)) \leq w(E_{G_i}(S^*, V \setminus S^*))$ and $\min\{\mathbf{d}_H(S^*_H), \mathbf{d}_H(K \setminus S^*_H)\} \geq \mathbf{d}(S^*)/3$, we have $\Psi^{\mathbf{d}_H}_H(S^*_H) \leq 3\Psi^{\mathbf{d}}_{G_i}(S^*) \leq 3\beta \cdot \psi^*$. Since H' is an α-approximate spectral sparsifier of H, we have $\Psi^{\mathbf{d}_H}_{H'}(S^*_H) \leq \alpha \cdot 3\Psi^{\mathbf{d}_H}_H(S^*_H) \leq 3\alpha\beta \cdot \psi^*$. By induction on the smaller recursive instance `WeightedBalCut`$(H', \mathbf{d}_H, \psi/\alpha, 3\alpha\beta\psi^*, b/3)$, the cut S'_H computed is a $(3\alpha\beta\psi^*, b/3)$-most-balanced $(\psi/\alpha, \mathbf{d}_H)$-sparse cut. Since H' is an α-approximate spectral sparsifier of H, we have $\Psi^{\mathbf{d}_H}_H(S'_H) \leq \alpha \cdot \Psi^{\mathbf{d}_H}_{H'}(S'_H) \leq \alpha \cdot \psi/\alpha = \psi$. Let S_H be the cut in G_i corresponding to S'_H after the uncontraction in Step Item 4d. By construction, $\Psi^{\mathbf{d}}_{G_i}(S_H) = \Psi^{\mathbf{d}_H}_H(S'_H) \leq \psi$ and $\mathbf{d}(S_H) = \mathbf{d}_H(S'_H)$. Since S^*_H is a cut with $\Psi^{\mathbf{d}_H}_{H'}(S^*_H) \leq 3\alpha\beta\psi^*$, we have
$$\mathbf{d}(S_H) = \mathbf{d}_H(S'_H) \geq \frac{\min\{\mathbf{d}_H(S^*_H), \mathbf{d}_H(K \setminus S^*_H)\}}{b/3} \geq \frac{\mathbf{d}(S^*)/3}{b/3} = \frac{\mathbf{d}(S^*)}{b}.$$

In both cases, the computed cut is a $(\psi/\psi^*, b)$-most-balanced ψ-sparse cut. □

The lemma below will be useful in bounding the running time of the recursive algorithm.

Lemma 8.7.11

For any integer $t \geq 1$ (as defined by the algorithm), the algorithm makes t recursive calls `WeightedBalCut`$(H', \mathbf{d}_H, \psi/\alpha, 3\alpha\beta\psi^*, b/3)$ on graphs H' with $\tilde{O}(\frac{m \log U}{t})$ vertices and $\tilde{O}(\frac{m \log^2 U}{t})$ edges, and runs in $\tilde{O}(tm)$ time outside these recursive calls.

Proof. By Lemma 8.2.7, computing the graphs $G_1, \ldots, G_t$ takes $\tilde{O}(tm)$ time. By Lemma 8.7.10, `RootedTreeBalCut` runs in $O(m)$ time for each G_t, for a total of $O(tm)$ time. Since each graph G_i is a $\tilde{O}(\frac{m \log U}{t})$-tree, by construction, each graph H_i has at most $\tilde{O}(\frac{m \log U}{t})$ vertices. By Corollary 8.6.2, the sparsified graphs H'_i have at most $\tilde{O}(\frac{m \log U}{t}) \log m \log U \leq \tilde{O}(\frac{m \log^2 U}{t})$ edges. □

Finally, we plug in our value $t = \lceil m^{1/r} (\log m)^{O(1)} \log^2 U \rceil$ that balances out the running time $\tilde{O}(tm)$ outside the recursive calls and the number r of recursion levels.

Theorem 8.7.12: Deterministic weighted most-balanced cut

Fix parameters $\psi^* > 0$ and $1 \leq r \leq O(\log m)$, and let $\psi = 12\beta \cdot (3\alpha^2\beta)^r \cdot \psi^*$. There is a deterministic algorithm that, given a weighted graph G with m edges and capacity ratio U and demands $\mathbf{d}$, computes a $(12\beta \cdot (3\alpha^2\beta)^r, 6 \cdot 3^r)$-most-balanced ψ-sparse cut in time $m^{1+1/r} (\log(mU))^{O(1)}$. Note that $12\beta \cdot (3\alpha^2\beta)^r = (\log m)^{O(r^3)}$.

Proof. Let G be the original graph with $m = m_0$ edges. Let G' be the current input graph in a recursive call of `WeightedBalCut`, with m' edges and capacity ratio U'. Set the parameters $t = \lceil m^{1/r}(\log m')^{O(1)} \log^2 U' \rceil$ from the algorithm and $\alpha = (\log m')^{O(r^2)}$ from Corollary 8.6.2 and $\beta = O(\log m'(\log\log m')^{O(1)}) \leq (\log m')^{O(1)}$ from Lemma 8.2.7. By Lemma 8.7.11, the algorithm makes $t = m^{1/r}(\log m')^{O(1)} \log^2 U'$ many recursive calls to graphs with at most $\tilde{O}(\frac{m' \log^2 U'}{t}) \leq m'/m^{1/r}$ edges, where U' is the capacity ratio of the current graph, so there are r levels of recursion. By Lemma 8.2.7, the capacity ratio of the graph increases by an $O(m)$ factor in each recursive call, so we have $U' \leq O(m)^r U$ for all recursive graphs, which means $t \leq m^{1/r}(r \log m + \log U)^{O(1)}$. By Lemma 8.7.11, the running time $\tilde{O}(tm')$ outside the recursive calls for this graph is $m'm^{1/r}(r \log m + \log U)^{O(1)}$. For recursion level $1 \leq i \leq r$, there are $m^{i/r} (r \log m + \log U)^{O(i)}$ many graphs at this recursion level, each with $m' \leq m^{1-i/r}$, so the total time spent on graphs at this level, outside their own recursive calls, is at most

$$m^{i/r} (r \log m + \log U)^{O(i)} \cdot m^{1-i/r} m^{1/r} (r \log m + \log U)^{O(1)} = m^{1+1/r} (r \log m + \log U)^{O(i)}.$$

Summed over all $1 \leq i \leq r$ and using $r \leq O(\log m)$, the overall total running time becomes $m^{1+1/r}(\log(mU))^{O(r)}$.

We also need to verify that the conditions $\psi \geq 12\beta \cdot \psi^*$ and $b \geq 6$ of Lemma 8.7.8 are always satisfied throughout the recursive calls. Since each recursive call decreases the parameter b by a factor of 3, and $b = 6 \cdot 3^r$ initially, the value of b is always at least 6. Also, in each recursive call, the ratio ψ/ψ^* decreases by a factor $3\alpha^2\beta$, so for the initial value $\psi = 12\beta \cdot (3\alpha^2\beta)^r \cdot \psi^*$ in the theorem statement, we always have $\psi/\psi^* \geq 12\beta$. □

8.7.3 Completing the Proof of Theorem 8.7.5 and Theorem 8.7.3

The proofs in this section follow the template from [87] but generalize it to work in weighted graphs and general demand. In order to prove Theorem 8.7.5, we first present the lemma below. Roughly, it guarantee the following. Given a set V' where $G[V']$ is "close" to being an expander in the sense that any sparse cut (A, B) in V' must be unbalanced: $\min\{\mathbf{d}(A), \mathbf{d}(B)\} \leq z$, then the algorithm returns a large subset $Y \subseteq V'$ such that Y is "closer" to being an expander. That is, any sparse cut (A', B') in Y must be even more unbalanced: $\min\{\mathbf{d}(A'), \mathbf{d}(B')\} \leq z' \ll z$.

Lemma 8.7.13

Let $G = (V, E)$ be a weighted graph with edge weights in $[1, U]$, and demands $\mathbf{d}(v) \in \{0\} \cup [1, U]$ for all $v \in V$ that are not all zero. There is a universal constant $c_1 > 0$ and a deterministic algorithm, that, given a vertex subset $V' \subseteq V$ with $\mathbf{d}(V') \geq \mathbf{d}(V)/2$, and parameters $r \geq 1$, $0 < \psi < 1$, $0 < z' < z$, such that for every partition (A, B) of V' with $w(E_G(A, B)) \leq \psi \cdot \min\{\mathbf{d}(A), \mathbf{d}(B)\}$, $\min\{\mathbf{d}(A), \mathbf{d}(B)\} \leq z$ holds, computes a partition (X, Y) of V', where $\mathbf{d}(X) \leq \mathbf{d}(Y)$ (where possibly $X = \emptyset$), $w(E_G(X, Y)) \leq \psi \cdot \mathbf{d}(X)$, and one of the following holds:

1. either $\mathbf{d}(X), \mathbf{d}(Y) \geq \mathbf{d}(V')/3$ (note that this can only happen if $z \geq \mathbf{d}(V')/3$); or
2. for every partition (A', B') of the set Y of vertices with
$$w(E_G(A', B')) \leq \frac{\psi}{(\log(mU))^{c_1 r^3}} \cdot \min\{\mathbf{d}(A'), \mathbf{d}(B')\},$$
$\min\{\mathbf{d}(A'), \mathbf{d}(B')\} \leq z'$ must hold (if $z' < 1$, then graph $G[Y]$ is guaranteed to have $\mathbf{d}$-sparsity at least $\psi/(\log(mU))^{c_1 r^3}$).

The running time of the algorithm is $O\left(\frac{z}{z'} \cdot m^{1+1/r} (\log(mU))^{O(1)}\right)$.

Proof. Our algorithm is iterative. At the beginning of iteration i, we are given a subgraph $G_i \subseteq G$, such that $\mathbf{d}(V(G_i)) \geq 2\mathbf{d}(V')/3$; at the beginning of the first iteration, we set $G_1 = G[V']$. At the end of iteration i, we either terminate the algorithm with the desired solution, or we compute a subset $S_i \subseteq V(G_i)$ of vertices, such that $\mathbf{d}(S_i) \leq \mathbf{d}(V(G_i))/2$, and $w(E_{G_i}(S_i, V(G_i) \setminus S_i)) \leq \psi \cdot \mathbf{d}(S_i)/2$. We then delete the vertices of S_i from G_i, in order to obtain the graph G_{i+1}, that serves as the input to the next iteration. The algorithm terminates once the current graph G_i satisfies $\mathbf{d}(V(G_i)) < 2\mathbf{d}(V')/3$ (unless it terminates with the desired output beforehand).

We now describe the execution of the ith iteration. We assume that the sets $S_1, \ldots, S_{i-1}$ of vertices are already computed, and that $\sum_{i'=1}^{i-1} \mathbf{d}(S_{i'}) \leq \mathbf{d}(V')/3$. Recall that G_i is the subgraph of $G[V']$ that is obtained by deleting the vertices of $S_1, \ldots, S_{i-1}$ from it. Recall also that we are guaranteed that $\mathbf{d}(V(G_i)) \geq 2\mathbf{d}(V')/3 \geq \mathbf{d}(V)/3$. We apply Theorem 8.7.12 to graph G_i with parameters $\psi^* = (\psi/2)/(\log(mU))^{c_1 r^3}$ and r, and let X be the returned set, which is a $(\psi^*, 6 \cdot 3^r)$-most-balanced $((\log(mU))^{c_1 r^3} \cdot \psi^*, \mathbf{d})$-sparse cut satisfying $\mathbf{d}(X) \leq \mathbf{d}(V)/2$.

We set parameter $z^* = z'/(6 \cdot 3^r)$. If $\mathbf{d}(X) \leq z^*$, then we terminate the algorithm, and return the partition (X, Y) of V' where $X = \bigcup_{i'=1}^{i} S_{i'}$, and $Y = V' \setminus X$. This satisfies the second condition of Lemma 8.7.13, since by the most-balanced sparse cut definition, every partition (A', B') of the set Y of vertices with $w(E_{\hat{G}}(A', B')) \leq \frac{\psi}{(\log(mU))^{c_1 r^3}} \cdot \min\{\mathbf{d}(A'), \mathbf{d}(B')\}$ must satisfy $\min\{\mathbf{d}(A'), \mathbf{d}(B')\} \leq 6 \cdot 3^r \cdot \mathbf{d}(X) < 6 \cdot 3^r \cdot z^* = z'$.

Otherwise, $\mathbf{d}(X) > z^*$. In this case, we set $S_i = X$ and continue the algorithm. If $\sum_{i'=1}^{i} \mathbf{d}(S_{i'}) \leq \mathbf{d}(V')/3$ continues to hold, then we let $G_{i+1} = G_i \setminus S_i$, and continue to the next iteration. Otherwise, we terminate the algorithm, and return the partition (X, Y) of V' where $X = \bigcup_{i'=1}^{i} S_{i'}$, and $Y = V' \setminus X$. Recall that we are guaranteed that $\mathbf{d}(X) \geq \mathbf{d}(V')/3$.

To show that $w(E_{\hat{G}}(X, Y)) \leq \psi \cdot \mathbf{d}(X)$, note that every cut S_i satisfies $w(E_{G_i}(S_i, V(G_i) \setminus S_i)) \leq (\psi/2)\mathbf{d}(S_i)$, so $w(E_G(X, Y)) \leq \sum_{i'=1}^{i} w(E_{G_i}(S_i, V(G_i) \setminus S_i)) \leq (\psi/2) \sum_{i'=1}^{i} \mathbf{d}(S_i) = (\psi/2)\mathbf{d}(X)$, which is at most $\psi \min\{\mathbf{d}(X), \mathbf{d}(V \setminus X)\}$ since $\mathbf{d}(X) \leq 2\mathbf{d}(V)/3$.

The bound on the running time of the algorithm proceeds similarly. Observe that we are guaranteed that for all i, $\mathbf{d}(S_i) \geq z^*$. Notice however that throughout the algorithm, if we set $A = \bigcup_{i'=1}^{i} S_{i'}$ and $B = V' \setminus A$, then $\mathbf{d}(A) < \mathbf{d}(B)$ holds, and $w(E_G(A, B)) \leq \psi \cdot \mathbf{d}(A)$. Therefore, from the condition of the lemma, $\mathbf{d}(A) \leq z$ must hold. Overall, the number of iterations in the algorithm is bounded by $z/z^* = 6 \cdot 3^r \cdot z/z'$, and, since every iteration takes time $m^{1+1/r} (\log(mU))^{O(1)}$, total running time of the algorithm is bounded by $\frac{z}{z'} \cdot m^{1+1/r} \cdot (\log(mU))^{O(1)}$. □

We are now ready to complete the proof of Theorem 8.7.5, which is almost identical to the proof of Theorem 7.5 of [27]. For completeness, we include the proof below.
Proof (Theorem 8.7.5). We first show that we can safely assume that $\mathbf{d}(V) \geq 2 \cdot 4^r$. Otherwise, consider the following expression in Item 2 of Lemma 8.7.13 and its upper bound:

$$\frac{\psi}{(\log(mU))^{c_1 r^3}} \cdot \min\{\mathbf{d}(A'), \mathbf{d}(B')\} \leq \frac{1}{(\log(mU))^{c_1 r^3}} \cdot 2 \cdot 4^r < 1,$$

which holds for large enough $c_1 > 0$. Since G is connected and all edges have weight at least 1, the condition in Item 2 only applies with $A' = \emptyset$ or $B' = \emptyset$. Therefore, the algorithm can trivially return $X = \emptyset$ and $Y = V$ and satisfy Item 2.

For the rest of the proof, assume that $\mathbf{d}(v) \geq 2 \cdot 4^r$. Our algorithm will consist of at most r iterations and uses the following parameters. First, we set $z_1 = \mathbf{d}(V)/2$, and for $1 < i \leq r$, we set $z_i = z_{i-1}/(\mathbf{d}(V)/2)^{1/r} \leq z_{i-1}/4$; in particular, $z_r = 1$ holds. We also define parameters $\psi_1, \ldots, \psi_r$, by letting $\psi_r = \psi$, and, for all $1 \leq i < r$, setting $\psi_i = 8 \cdot (\log(mU))^{c_1 r^3} \cdot \psi_{i+1}$, where c_1 is the constant from lemz. Notice that $\psi_1 \leq \psi \cdot (\log m)^{O(r^4)}$.

In the first iteration, we apply Lemma 8.7.13 to the set $V' = V$ of vertices, with the parameters $\psi = \psi_1$, $z = z_1$, and $z' = z_2$. Clearly, for every partition (A, B) of V' with $w_G(E_G(A, B)) \leq \psi_1 \cdot \min\{\mathbf{d}(A), \mathbf{d}(B)\}$, it holds that $\min\{\mathbf{d}(A), \mathbf{d}(B)\} \leq z_1 = \mathbf{d}(V)/2$. If the outcome of the algorithm from Lemma 8.7.13 is a partition (X, Y) of V satisfying $\mathbf{d}(X), \mathbf{d}(Y) \geq \mathbf{d}(V)/3$ and $w_G(E_G(X, Y)) \leq \psi_1 \cdot \min\{\mathbf{d}(X), \mathbf{d}(Y)\} \leq \psi \cdot (\log m)^{O(r^4)} \min\{\mathbf{d}(X), \mathbf{d}(Y)\}$, then we return the cut (X, Y) and terminate the algorithm.

We assume from now on that the algorithm from Lemma 8.7.13 returned a partition (X, Y) of V, where $\mathbf{d}(X) \leq \mathbf{d}(Y)$ (where possibly $X = \emptyset$), $\mathbf{d}(X) \leq \mathbf{d}(V)/3$, $w_G(E_G(X, Y)) \leq$

$\psi_1 \cdot \mathbf{d}(X)$, and the following guarantee holds: For every partition (A', B') of the set Y of vertices with $w_G(E_G(A', B')) \le 8\psi_2 \cdot \min\{\mathbf{d}(A'), \mathbf{d}(B')\}$, it holds that $\min\{\mathbf{d}(A'), \mathbf{d}(B')\} \le z_2$. We set $S_1 = X$, and we let $G_2 = G \setminus S_1$.

The remainder of the algorithm consists of $r-1$ iterations $i = 2, 3, \ldots, r$. The input to iteration i is a subgraph $G_i \subseteq G$ with $\mathbf{d}(V(G_i)) \le \mathbf{d}(V)/2$, such that for every cut (A', B') of G_i with $w_G(E_G(A', B')) \le \psi_i \cdot \min\{\mathbf{d}(A'), \mathbf{d}(B')\}$, it holds that $\min\{\mathbf{d}(A'), \mathbf{d}(B')\} \le z_i$. (Observe that, as established above, this condition holds for graph G_2). The output is a subset $S_i \subseteq V(G_i)$ of vertices, such that $\mathbf{d}(S_i) \le \mathbf{d}(V(G_i))/2$ and $w_G(E_{G_i}(S_i, V(G_i) \setminus S_i)) \le \psi_i \cdot \mathbf{d}(S_i)$, and, if we set $G_{i+1} = G_i \setminus S_i$, then we are guaranteed that for every cut (A'', B'') of G_{i+1} with $w_G(E_G(A'', B'')) \le 8\psi_{i+1} \cdot \min\{\mathbf{d}(A''), \mathbf{d}(B'')\}$, it holds that $\min\{\mathbf{d}(A''), \mathbf{d}(B'')\} \le z_{i+1}$. In particular, if $w_G(E_G(A'', B'')) \le \psi_{i+1} \cdot \min\{\mathbf{d}(A''), \mathbf{d}(B'')\}$, then $\min\{\mathbf{d}(A''), \mathbf{d}(B'')\} \le z_{i+1}$ holds. In order to execute the ith iteration, we simply apply Lemma 8.7.13 to the set $V' = V(G_i)$ of vertices, with parameters $\psi = \psi_i$, $z = z_i$ and $z' = z_{i+1}$. As we show later, we will ensure that $\mathbf{d}(V(G_i)) \ge \mathbf{d}(V)/2$. Since, for $i > 1$, $z_i \le \mathbf{d}(V)/8 < \mathbf{d}(V)/6 \le \mathbf{d}(V(G_i))/3$, the outcome of the lemma must be a partition (X, Y) of V', where $\mathbf{d}(X) \le \mathbf{d}(Y)$ (where possibly $X = \emptyset$), $w_G(E_G(X, Y)) \le \psi_i \cdot \mathbf{d}(X)$, and we are guaranteed that, for every partition (A'', B'') of the set Y of vertices with $w_G(E_G(A'', B'')) \le 8\psi_{i+1} \cdot \min\{\mathbf{d}(A''), \mathbf{d}(B'')\}$, it holds that $\min\{\mathbf{d}(A'), \mathbf{d}(B')\} \le z_{i+1}$. Therefore, we can simply set $S_i = X$, $G_{i+1} = G_i \setminus S_i$, and continue to the next iteration, provided that $\mathbf{d}(V(G_{i+1})) \ge \mathbf{d}(V)/2$ holds.

We next show that this indeed must be the case. Recall that for all $2 \le i' \le i$, we guarantee that $\mathbf{d}(S_{i'}) \le z_{i'} \le \mathbf{d}(V)/(2 \cdot 4^{i'-1})$. Therefore, if we denote by $Z = \bigcup_{i'=2}^{i} S_{i'}$ and $Z' = V(G_2) \setminus Z$, then $\mathbf{d}(Z) \le \mathbf{d}(V)/2 \cdot \sum_{i'=2}^{i} 1/4^{i'-1} \le \mathbf{d}(V)/6$, so

$$\mathbf{d}(V(G_{i+1})) = \mathbf{d}(Z') = \mathbf{d}(V(G_2)) - \mathbf{d}(Z) \ge 2\mathbf{d}(V)/3 - \mathbf{d}(V)/6 = \mathbf{d}(V)/2.$$

as promised.

We continue the algorithm until we reach the last iteration, where $z_r = 1$ holds. Apply Lemma 8.7.13 to the final graph G_r with $z' = 1/2$ to obtain $S_r \subseteq V(G_r)$. Since $z' < 1$, the discussion in Item 2 implies that graph $G_r \setminus S_r$ has $\mathbf{d}$-sparsity at least ψ (recall that $\psi_r = \psi$). We define our final partition as $Y = V(G_r) \setminus S_r$ and $X = V \setminus Y = \bigcup_{i=1}^{r} S_i$. By the same reasoning as before, we are guaranteed that $\mathbf{d}(Y) \ge \mathbf{d}(V)/2 \ge \mathbf{d}(X)$. Finally,

$$w_G(E_G(X, Y)) \le \sum_{i=1}^{r} w_G(E_G(S_i, V(G_i) \setminus S_i)) \le \sum_{i=1}^{r} \psi_i \cdot \mathbf{d}(S_i) \le \psi \cdot (\log m)^{O(r^4)} \cdot \mathbf{d}(X),$$

which concludes the proof of Theorem 8.7.5. $\square$

Finally, we prove Theorem 8.7.3, which is almost identical to the proof of Theorem 8.6.1.
Proof (Theorem 8.7.3). We maintain a collection $\mathcal{H}$ of disjoint sub-graphs of G that we call

clusters, which is partitioned into two subsets, set $\mathcal{H}^A$ of *active clusters*, and set $\mathcal{H}^I$ of *inactive clusters*. We ensure that each inactive cluster $H \in \mathcal{H}^I$ has $\mathbf{d}|_{V(H)}$-sparsity at least ψ. We also maintain a set E' of "deleted" edges, that are not contained in any cluster in $\mathcal{H}$. At the beginning of the algorithm, we let $\mathcal{H} = \mathcal{H}^A = \{G\}$, $\mathcal{H}^I = \emptyset$, and $E' = \emptyset$. The algorithm proceeds as long as $\mathcal{H}^A \neq \emptyset$, and consists of iterations. For convenience, we denote $\alpha = (\log m)^{O(r^4)}$ the approximation factor achieved by the algorithm from Theorem 8.7.5, and we set $\psi = \epsilon/(c\alpha \cdot \log(mU))$, for some large enough constant c, so that $\psi = \Omega\left(\epsilon/\big(\log^{O(r^4)} m \log U\big)\right)$ holds.

In every iteration, we apply the algorithm from Theorem 8.7.5 to every graph $H \in \mathcal{H}^A$, with the same parameters α, r, and ψ. Consider the partition (A, B) of $V(H)$ that the algorithm computes, with $w(E_H(A, B)) \leq \alpha\psi \cdot \mathbf{d}(V(H)) \leq \frac{\epsilon \cdot \mathbf{d}(V(H))}{c \log(mU)}$. We add the edges of $E_H(A, B)$ to set E'. If $\mathbf{d}(A), \mathbf{d}(B) \geq \mathbf{d}(V(H))/3$, then we replace H with $H[A]$ and $H[B]$ in $\mathcal{H}$ and in $\mathcal{H}^A$. Otherwise, we are guaranteed that $\mathbf{d}(A) \geq \mathbf{d}(V(H))/2$ and $\Psi^{\mathbf{d}|_A}(H[A]) \geq \psi$. Then we remove H from $\mathcal{H}$ and $\mathcal{H}^A$, add $H[A]$ to $\mathcal{H}$ and $\mathcal{H}^I$, and add $H[B]$ to $\mathcal{H}$ and $\mathcal{H}^A$.

When the algorithm terminates, $\mathcal{H}^A = \emptyset$, and so every graph $H \in \mathcal{H}$ has $\mathbf{d}|_{V(H)}$-sparsity at least ψ. Notice that in every iteration, the maximum value of $\mathbf{d}(V(H))$ of a graph $H \in \mathcal{H}^A$ must decrease by a constant factor. Therefore, the number of iterations is bounded by $O(\log(mU))$. It is easy to verify that the total weight of edges added to set E' in every iteration is at most $\frac{\epsilon \cdot \mathbf{d}(V)}{c \log(mU)}$. Therefore, by letting c be a large enough constant, we can ensure that $w(E') \leq \epsilon \mathbf{d}(V)$. The output of the algorithm is the partition $\mathcal{P} = \{V(H) \mid H \in \mathcal{H}\}$ of V. From the above discussion, we obtain a valid (ϵ, ψ)-expander decomposition, for $\psi = \Omega\left(\epsilon/\big(\log^{O(r^4)} m \log U\big)\right)$.

It remains to analyze the running time of the algorithm. The running time of a single iteration is bounded by $m \cdot (mU)^{O(1/r)}$. Since the total number of iterations is bounded by $O(\log(mU))$, we get that the total running time of the algorithm is $m \cdot (mU)^{O(1/r)} \log(mU)$. □

8.8 Weighted Expander Decomposition, Boundary-Linked

In this section, we augment the WeightedBalCutPrunealgorithm (Theorem 8.7.5) to handle an additional *boundary-linkedness* property, though we restrict to standard vertex demands where the demand of each vertex is its (weighted) degree. Our proof is directly modeled off of the proof of Theorem 4.5 in [45], so we claim no novelty in this section.

For simplicity, we will work with weighted multigraphs with *self-loops*, and we re-define the degree $\deg(v)$ to mean $w(\partial(\{v\}))$ plus the total weight of all self-loops at vertex v. All other definitions that depend on $\deg(v)$, such as $\mathbf{vol}(S)$ and $\Phi(G)$, are also affected.

Given a weighted graph $G = (V, E)$, a parameter $r > 0$, and a subset $A \subseteq V$, define $G\{A\}^r$ as the graph $G[A]$ with the following self-loops attached: for each edge $e \in E(A, V \setminus A)$ with endpoint $v \in A$, add a self-loop at v of weight $r \cdot w(e)$.

We now present the formal definition of boundary-linked expander decomposition.

Definition 8.8.1: Boundary-linked expander decomposition

Let $G = (V, E)$ be a graph and let $r \geq 1$ be a parameter. A β-boundary-linked ϕ-expander decomposition is a partition $V = V_1 \uplus \cdots \uplus V_k$ of V such that

1. For each i, $G[V_i]^{\beta/\phi}$ is a ϕ-expander. In particular, for any S satisfying
$$\mathbf{vol}_{G[V_i]}(S) + \frac{\beta}{\phi} w(E_G(S, V \setminus V_i)) \leq \mathbf{vol}_{G[V_i]}(V_i \setminus S) + \frac{\beta}{\phi} w(E_G(V_i \setminus S, V \setminus V_i)),$$
we simultaneously obtain
$$\frac{w(\partial_{G[V_i]} S)}{\mathbf{vol}_{G[V_i]}(S)} \geq \phi \quad \text{and} \quad \frac{w(\partial_{G[V_i]} S)}{\frac{\beta}{\phi} w(E_G(S, V \setminus V_i))} \geq \phi \iff \frac{w(\partial_{G[V_i]} S)}{w(E_G(S, V \setminus V_i))} \geq \beta.$$
The right-most inequality is where the name "boundary-linked" comes from.
2. The total weight of "inter-cluster" edges, $w(\partial V_1 \cup \cdots \cup \partial V_k)$, is at most $(\log n)^{O(r^4)} \phi \mathbf{vol}(V)$.

The main result of this section is an algorithm for this decomposition.

Theorem 8.8.2: Boundary-linked expander decomposition

For any parameters $\beta \leq (\log n)^{-O(r^4)}$ and $\phi \leq \beta$, there is a deterministic β-boundary-linked ϕ-expander decomposition algorithm that runs in time $m^{1+O(1/r)} + \tilde{O}(m/\phi^2)$,

Our algorithm uses the WeightedBalCutPrunealgorithm (Definition 8.7.4) from Section 8.7 with one simple modification in the **(Prune)** case. The only new ingredient we need is an additional *trimming* step described in the lemma below. While [45] prove it for unweighted graphs only, the algorithm translates directly to the weighted case;[2] see, for example, Theorem 4.2 of [94].

Lemma 8.8.3: Trimming, Lemmas 4.9 and 4.10 of [45]

Given a weighted graph $G = (V, E)$ and subset $A \subseteq V$ such that $G\{A\}$ is an 8ϕ-expander and $w(E_G(A, V \setminus A)) \leq \frac{\phi}{16} \mathbf{vol}_G(A)$, we can compute a "pruned" set $P \subseteq A$ in deterministic $\tilde{O}(m/\phi^2)$ time with the following properties:

1. $\mathbf{vol}_G(P) \leq \frac{4}{\phi} w(E_G(A, V \setminus A))$,
2. $w(E_G(A', V \setminus A')) \leq 2w(E_G(A, V \setminus A))$ where $A' := A \setminus P$, and
3. $G\{A'\}^{1/(8\phi)}$ is a ϕ-expander.

[2]In particular, the core subroutine, called *Unit-Flow* in [94], is based on the push-relabel max-flow algorithm, which works on both unweighted and weighted graphs.

We now prove Theorem 8.8.2 using Theorem 8.7.5. Our proof is copied almost ad verbatim from the proof of Theorem 8.6.1, with the necessary changes to prove the additional boundary-linked property.

We maintain a collection $\mathcal{H}$ of vertex-disjoint graphs that we call *clusters*, which are subgraphs of G with some additional self-loops. The set $\mathcal{H}$ of clusters is partitioned into two subsets, set $\mathcal{H}^A$ of *active clusters*, and set $\mathcal{H}^I$ of *inactive clusters*. We ensure that each inactive cluster $H \in \mathcal{H}^I$ is a ϕ-expander. We also maintain a set E' of "deleted" edges, that are not contained in any cluster in $\mathcal{H}$. At the beginning of the algorithm, we let $\mathcal{H} = \mathcal{H}^A = \{G\}$, $\mathcal{H}^I = \emptyset$, and $E' = \emptyset$. The algorithm proceeds as long as $\mathcal{H}^A \neq \emptyset$, and consists of iterations. Let $\alpha = (\log n)^{O(r^4)}$ be the approximation factor from Theorem 8.7.5.

In every iteration, we apply the algorithm from Theorem 8.7.5 to every graph $H \in \mathcal{H}^A$, with the same parameters α, r, and ϕ. Let U be the vertices of H. Consider the cut (A, B) in H that the algorithm returns, with

$$w(E_H(A,B)) \le \alpha\phi \cdot \mathbf{vol}(U) \le \frac{\epsilon \cdot \mathbf{vol}(U)}{c \log n}. \tag{8.5}$$

We add the edges of $E_H(A,B)$ to set E'.

If $\mathbf{vol}_H(B) \ge \frac{\mathbf{vol}(U)}{32\alpha}$, then we replace H with $H\{A\}^{1/(\alpha^2\phi\log n)}$ and $H\{B\}^{1/(\alpha^2\phi\log n)}$ in $\mathcal{H}$ and in $\mathcal{H}^A$. Note that the self-loops add a total volume of

$$\frac{1}{\alpha^2\phi\log n} \cdot w(E_H(A,B)) \le \frac{1}{\alpha^2\phi\log n} \cdot \alpha\phi\,\mathbf{vol}(U) = \frac{1}{\alpha\log n}\mathbf{vol}(U). \tag{8.6}$$

Otherwise, if $\mathbf{vol}_H(B) < \frac{\mathbf{vol}(U)}{32\alpha} \le \mathbf{vol}(U)/3$, then we must be in the **(Prune)** case, which means that $\mathbf{vol}_H(A) \ge \mathbf{vol}(U)/2$ and graph $H\{A\}^{1/(8\phi)}$ has conductance at least ϕ. Since

$$w(E_H(A,B)) \le \alpha\phi \cdot \mathbf{vol}_H(B) \le \frac{\phi}{32}\mathbf{vol}(U) \le \frac{\phi}{16}\mathbf{vol}(A),$$

we can call Lemma 8.8.3 on A to obtain a pruned set $P \subseteq A$ such that

$$\mathbf{vol}_H(P) \le \frac{4}{\phi} w(E_H(A,B)) \le \frac{1}{8}\mathbf{vol}(U)$$

and

$$w(E_H(A', U \setminus A')) \le 2w(E_H(A,B)) \le \frac{\phi}{8}\mathbf{vol}(A)$$

for $A' := A \setminus P$, and $H\{A'\}^{1/(8\phi)}$ is a ϕ-expander. Add the edges of $E_H(A', U \setminus A')$ to E', remove H from $\mathcal{H}$ and $\mathcal{H}^A$, add $H\{A'\}^{1/(8\phi)}$ to $\mathcal{H}$ and $\mathcal{H}^I$, and add $H\{B \cup P\}^{1/(8\phi)}$ to $\mathcal{H}$ and $\mathcal{H}^A$. Observe that

$$\mathbf{vol}_H(B \cup P) = \mathbf{vol}_H(B) + \mathbf{vol}_H(P) \le \frac{1}{2}\mathbf{vol}_H(U) + \frac{1}{8}\mathbf{vol}_H(U) \le \frac{5}{8}\mathbf{vol}(U).$$

When the algorithm terminates, $\mathcal{H}^A = \emptyset$, and so every graph in $\mathcal{H}$ has conductance at least ϕ. Notice that in every iteration, the maximum volume of a graph in $\mathcal{H}^A$ is at most a factor $(1 - \frac{1}{32\alpha})$ of what it was before. Since edge weights are polynomially bounded, the number of iterations is at most $O(\alpha \log n)$. On each iteration, the total volume of graphs in $\mathcal{H}^A$ increases by at most factor $1 + \frac{2}{\alpha \log n}$ factor due to the self-loops added in (8.6), so the total volume of all $H \in \mathcal{H}$ at the end is at most a constant factor of the initial volume $\mathbf{vol}_G(V)$.

The output of the algorithm is the partition of V induced by the vertex sets of $H \in \mathcal{H}$, so the inter-cluster edges is a subset of E'. It is easy to verify by (8.5) that the total weight of edges added to set E' in every iteration is at most $\alpha\phi$ times the total volume of graphs in $\mathcal{H}^A$ at the beginning of that iteration, which is $O(\mathbf{vol}_G(V))$. Over all $O(\alpha \log n)$ iterations, the total weight of E' is $O(\alpha \log n) \cdot \alpha\phi \, \mathbf{vol}_G(V) \leq (\log n)^{O(r^4)} \phi \mathbf{vol}_G(V)$, fulfilling property (2) of a boundary-linked expander decomposition.

It remains to show that for each graph $H \in \mathcal{H}^I$, its vertex set U satisfies the boundary-linked ϕ-expander property (1) of Definition 8.8.1. For each boundary edge $e \in E_G(U, V \setminus U)$, it was created at some iteration where we either added $\frac{1}{\alpha^2 \phi \log n}$ self-loops or $\frac{1}{8\phi}$ self-loops, so $G[U]^{\min\{1/(\alpha^2\phi \log n), 1/(8\phi)\}}$ is a subgraph of H. Since H is a ϕ-expander, so is $G[U]^{\min\{1/(\alpha^2\phi \log n), 1/(8\phi)\}}$, and property (1) for $\beta := \min\{1/\alpha^2, 1/8\}$ follows.

It remains to analyze the running time of the algorithm. The running time of a single iteration is bounded by $O(m^{1+O(1/r)}) + \tilde{O}(m/\phi^2)$. Since the total number of iterations is bounded by $O(\log n)$, the total running time is the same, asymptotically.

8.9 Conclusion

In this chapter, we presented a deterministic, almost-linear time algorithm for various settings of expander decomposition, which opened the door to the fast, deterministic, preconditioning-based algorithms of Chapters 3 and 6. One immediate open problem is whether the $n^{o(1)}$ factors in the running time and expander decomposition quality can be improved to polylog(n), which can indeed be done in the randomized case. Achieving such a result deterministically will likely require substantially new ideas that avoid the recursive nature of our approach. We remark that even a Las Vegas expander decomposition algorithm that achieves polylog(n) factors everywhere is still unknown.

Bibliography

[1] Amir Abboud, Greg Bodwin, and Seth Pettie. A hierarchy of lower bounds for sublinear additive spanners. *SIAM Journal on Computing*, 47(6):2203–2236, 2018.

[2] Amir Abboud, Robert Krauthgamer, and Ohad Trabelsi. New algorithms and lower bounds for all-pairs max-flow in undirected graphs. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 48–61. SIAM, 2020.

[3] Amir Abboud, Robert Krauthgamer, and Ohad Trabelsi. Cut-equivalent trees are optimal for min-cut queries. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020*. IEEE Computer Society, 2020.

[4] Ittai Abraham and Ofer Neiman. Using petal-decompositions to build a low stretch spanning tree. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 395–406. ACM, 2012.

[5] Zeyuan Allen Zhu, Zhenyu Liao, and Lorenzo Orecchia. Spectral sparsification and regret minimization beyond matrix multiplicative updates. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 237–245, 2015.

[6] Noga Alon. Eigenvalues and expanders. *Combinatorica*, 6(2):83–96, 1986. doi: 10.1007/BF02579166. URL `https://doi.org/10.1007/BF02579166`.

[7] Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *Journal of the ACM (JACM)*, 42(4):844–856, 1995.

[8] Reid Andersen, Fan R. K. Chung, and Kevin J. Lang. Using pagerank to locally partition a graph. *Internet Mathematics*, 4(1):35–64, 2007. doi: 10.1080/15427951.2007.10129139. URL `https://doi.org/10.1080/15427951.2007.10129139`.

[9] Alexandr Andoni, Clifford Stein, and Peilin Zhong. Parallel approximate undirected shortest paths via low hop emulators. *arXiv preprint arXiv:1911.01956*, 2019.

[10] Sanjeev Arora, Satish Rao, and Umesh V. Vazirani. Expander flows, geometric embeddings and graph partitioning. *J. ACM*, 56(2):5:1–5:37, 2009. doi: 10.1145/1502793.1502794. URL `https://doi.org/10.1145/1502793.1502794`.

[11] Sanjeev Arora, Elad Hazan, and Satyen Kale. $O(\sqrt{\log n})$ approximation to SPARS-

EST CUT in $\widetilde{O}(n^2)$ time. *SIAM J. Comput.*, 39(5):1748–1771, 2010. doi: 10.1137/080731049. URL `https://doi.org/10.1137/080731049`.

[12] Baruch Awerbuch, Andrew V Goldberg, Michael Luby, and Serge A Plotkin. Network decomposition and locality in distributed computation. In *FOCS*, volume 30, pages 364–369, 1989.

[13] Joshua Batson, Daniel A Spielman, and Nikhil Srivastava. Twice-Ramanujan sparsifiers. *SIAM Journal on Computing*, 41(6):1704–1721, 2012.

[14] Ruben Becker, Andreas Karrenbauer, Sebastian Krinninger, and Christoph Lenzen. Near-optimal approximate shortest paths and transshipment in distributed and streaming models. *arXiv preprint arXiv:1607.05127*, 2016.

[15] András A Benczúr and David R Karger. Approximate s-t min-cuts in $\tilde{o}(n^2)$ time. *SIAM Journal on Computing*, 44(2):290–319, 2015.

[16] Anand Bhalgat, Ramesh Hariharan, Telikepalli Kavitha, and Debmalya Panigrahi. An õ(mn) gomory-hu tree construction algorithm for unweighted graphs. In David S. Johnson and Uriel Feige, editors, *Proceedings of the 39th Annual ACM Symposium on Theory of Computing, San Diego, California, USA, June 11-13, 2007*, pages 605–614. ACM, 2007.

[17] Anand Bhalgat, Ramesh Hariharan, Telikepalli Kavitha, and Debmalya Panigrahi. Fast edge splitting and edmonds' arborescence construction for unweighted graphs. In Shang-Hua Teng, editor, *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2008, San Francisco, California, USA, January 20-22, 2008*, pages 455–464, 2008.

[18] Ruoxu Cen, Jason Li, Danupon Nanongkai, Debmalya Panigrahi, and Thatchaphol Saranurak. Minimum cuts in directed graphs via $\sqrt{n}$ max-flows. *arXiv preprint arXiv:2104.07898*, 2021.

[19] Ruoxu Cen, Jason Li, and Debmalya Panigrahi. Augmenting edge connectivity via isolating cuts. 2021.

[20] Yi-Jun Chang and Thatchaphol Saranurak. Improved distributed expander decomposition and nearly optimal triangle enumeration. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019.*, pages 66–73, 2019. doi: 10.1145/3293611.3331618. URL `https://doi.org/10.1145/3293611.3331618`.

[21] Chandra Chekuri and Julia Chuzhoy. Large-treewidth graph decompositions and applications. In *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 291–300, 2013. doi: 10.1145/2488608.2488645. URL `https://doi.org/10.1145/2488608.2488645`.

[22] Chandra Chekuri and Julia Chuzhoy. Polynomial bounds for the grid-minor theorem. *J. ACM*, 63(5):40:1–40:65, 2016. doi: 10.1145/2820609. URL `https://doi.org/10.`

1145/2820609.

[23] Chandra Chekuri and Kent Quanrud. Isolating cuts,(bi-) submodularity, and faster algorithms for global connectivity problems. *arXiv preprint arXiv:2103.12908*, 2021.

[24] Timothy Chu, Yu Gao, Richard Peng, Sushant Sachdeva, Saurabh Sawlani, and Junxing Wang. Graph sparsification, spectral sketches, and faster resistance computation, via short cycle decompositions. In *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018*, pages 361–372, 2018. doi: 10.1109/FOCS.2018.00042. URL https://doi.org/10.1109/FOCS.2018.00042.

[25] Julia Chuzhoy and Sanjeev Khanna. A new algorithm for decremental single-source shortest paths with applications to vertex-capacitated flow and cut problems. In *STOC*, pages 389–400. ACM, 2019. To appear at STOC'19.

[26] Julia Chuzhoy and Shi Li. A polylogarithmic approximation algorithm for edge-disjoint paths with congestion 2. *J. ACM*, 63(5):45:1–45:51, 2016. doi: 10.1145/2893472. URL https://doi.org/10.1145/2893472.

[27] Julia Chuzhoy, Yu Gao, Jason Li, Danupon Nanongkai, Richard Peng, and Thatchaphol Saranurak. A deterministic algorithm for balanced cut with applications to dynamic connectivity, flows, and beyond. In *Symp. Foundations of Computer Science (FOCS)*, 11 2020.

[28] Edith Cohen. Polylog-time and near-linear work approximation scheme for undirected shortest paths. *Journal of the ACM (JACM)*, 47(1):132–166, 2000.

[29] Vincent Cohen-Addad, Anupam Gupta, Amit Kumar, Euiwoong Lee, and Jason Li. Tight fpt approximations for k-median and k-means. *arXiv preprint arXiv:1904.12334*, 2019.

[30] Vincent Cohen-Addad, Anupam Gupta, Philip N. Klein, and Jason Li. A quasipolynomial $(2 + \varepsilon)$-approximation for planar sparsest cut. *arXiv preprint arXiv:2105.15187*, 2021.

[31] Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized algorithms*. Springer, Cham, 2015. ISBN 978-3-319-21274-6; 978-3-319-21275-3. doi: 10.1007/978-3-319-21275-3. URL http://dx.doi.org/10.1007/978-3-319-21275-3.

[32] Yefim Dinitz. Dinitz' algorithm: The original version and Even's version. In *Theoretical computer science*, pages 218–240. Springer, 2006.

[33] Jack Edmonds. Edge-disjoint branchings. *Combinatorial algorithms*, 1973.

[34] Michael Elkin and Ofer Neiman. Efficient algorithms for constructing very sparse spanners and emulators. *ACM Transactions on Algorithms (TALG)*, 15(1):4, 2018.

[35] David J Evans. The use of pre-conditioning in iterative methods for solving linear equations with symmetric positive definite matrices. *IMA Journal of Applied Mathematics*,

4(3):295–314, 1968.

[36] Shimon Even and Yossi Shiloach. An on-line edge-deletion problem. *Journal of the ACM (JACM)*, 28(1):1–4, 1981.

[37] Jeremy T Fineman. Nearly work-efficient parallel algorithm for digraph reachability. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, pages 457–470. ACM, 2018.

[38] Ofer Gabber and Zvi Galil. Explicit constructions of linear-sized superconcentrators. *J. Comput. Syst. Sci.*, 22(3):407–420, 1981. announced at FOCS'79.

[39] Harold Gabow, Zvi Galil, Thomas Spencer, and Robert Tarjan. Efficient algorithms for finding minimum spanning tree in undirected and directed graphs. *Combinatorica*, 6:109–122, 06 1986. doi: 10.1007/BF02579168.

[40] H.N. Gabow. A matroid approach to finding edge connectivity and packing arborescences. *Journal of Computer and System Sciences*, 50(2):259–273, 1995. ISSN 0022-0000. doi: https://doi.org/10.1006/jcss.1995.1022.

[41] Yu Gao, Jason Li, Danupon Nanongkai, Richard Peng, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. Deterministic graph cuts in subquadratic time: Sparse, balanced, and k-vertex. *arXiv preprint arXiv:1910.07950*, 2019.

[42] Andrew V. Goldberg and Robert Endre Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988. doi: 10.1145/48014.61051. URL `https://doi.org/10.1145/48014.61051`.

[43] Oded Goldreich and Dana Ron. A sublinear bipartiteness tester for bounded degree graphs. *Combinatorica*, 19(3):335–373, 1999. doi: 10.1007/s004930050060. URL `https://doi.org/10.1007/s004930050060`.

[44] Ralph E Gomory and Tien Chung Hu. Multi-terminal network flows. *Journal of the Society for Industrial and Applied Mathematics*, 9(4):551–570, 1961.

[45] Gramoz Goranci, Harald Räcke, Thatchaphol Saranurak, and Zihan Tan. The expander hierarchy and its applications to dynamic graph algorithms. *arXiv preprint arXiv:2005.02369*, 2020.

[46] Anupam Gupta, Euiwoong Lee, and Jason Li. The Karger-Stein algorithm is optimal for k-cut. In *ACM Symposium on the Theory of Computing (STOC)*, 5 2020. doi: 10.1145/3357713.3384285. URL `https://doi.org/10.1145/3357713.3384285`.

[47] Anupam Gupta, Euiwoong Lee, and Jason Li. The connectivity threshold for dense graphs. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 89–105. SIAM, 2021.

[48] Jianxiu Hao and James B. Orlin. A faster algorithm for finding the minimum cut in a directed graph. *J. Algorithms*, 17(3):424–446, 1994. ISSN 0196-6774. doi: 10.1006/jagm.1994.1043. URL `http://dx.doi.org/10.1006/jagm.1994.1043`. Third Annual

ACM-SIAM Symposium on Discrete Algorithms (Orlando, FL, 1992).

[49] Sariel Har-Peled. *Geometric approximation algorithms*. Number 173. American Mathematical Soc., 2011.

[50] Ramesh Hariharan, Telikepalli Kavitha, and Debmalya Panigrahi. Efficient algorithms for computing all low s-t edge connectivities and related problems. In Nikhil Bansal, Kirk Pruhs, and Clifford Stein, editors, *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007*, pages 127–136. SIAM, 2007.

[51] Monika Henzinger, Satish Rao, and Di Wang. Local flow partitioning for faster edge connectivity. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 1919–1938, 2017. doi: 10.1137/1.9781611974782.125. URL `https://doi.org/10.1137/1.9781611974782.125`.

[52] William B Johnson and Joram Lindenstrauss. Extensions of lipschitz mappings into a hilbert space. *Contemporary mathematics*, 26(189-206):1, 1984.

[53] Ravi Kannan, Santosh Vempala, and Adrian Vetta. On clusterings: Good, bad and spectral. *J. ACM*, 51(3):497–515, 2004. doi: 10.1145/990308.990313. URL `https://doi.org/10.1145/990308.990313`.

[54] David R Karger. Global min-cuts in rnc, and other ramifications of a simple min-cut algorithm. In *SODA*, volume 93, pages 21–30, 1993.

[55] David R. Karger. Minimum cuts in near-linear time. *J. ACM*, 47(1):46–76, 2000. ISSN 0004-5411. doi: 10.1145/331605.331608. URL `http://dx.doi.org/10.1145/331605.331608`.

[56] David R Karger and Clifford Stein. A new approach to the minimum cut problem. *Journal of the ACM (JACM)*, 43(4):601–640, 1996.

[57] Ken-ichi Kawarabayashi and Mikkel Thorup. Deterministic edge connectivity in near-linear time. *J. ACM*, 66(1):4:1–4:50, 2019. doi: 10.1145/3274663. URL `https://doi.org/10.1145/3274663`.

[58] Rohit Khandekar, Subhash Khot, Lorenzo Orecchia, and Nisheeth K Vishnoi. On a cut-matching game for the sparsest cut problem. *Univ. California, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2007-177*, 2007.

[59] Rohit Khandekar, Satish Rao, and Umesh V. Vazirani. Graph partitioning using single commodity flows. *J. ACM*, 56(4):19:1–19:15, 2009. doi: 10.1145/1538902.1538903. URL `https://doi.org/10.1145/1538902.1538903`.

[60] Andrey Boris Khesin, Aleksandar Nikolov, and Dmitry Paramonov. Preconditioning for the geometric transportation problem. *arXiv preprint arXiv:1902.08384*, 2019.

[61] Philip N Klein and Sairam Subramanian. A parallel randomized approximation scheme

for shortest paths. In *STOC*, volume 92, pages 750–758, 1992.

[62] Philip N Klein and Sairam Subramanian. A randomized parallel algorithm for single-source shortest paths. *Journal of Algorithms*, 25(2):205–220, 1997.

[63] Ioannis Koutis, Gary L Miller, and Richard Peng. Approaching optimality for solving sdd linear systems. *SIAM Journal on Computing*, 43(1):337–354, 2014.

[64] Rasmus Kyng, Yin Tat Lee, Richard Peng, Sushant Sachdeva, and Daniel A. Spielman. Sparsified cholesky and multigrid solvers for connection laplacians. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 842–850, 2016. doi: 10.1145/2897518.2897640. URL `https://doi.org/10.1145/2897518.2897640`.

[65] Yin Tat Lee and Aaron 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, Philadelphia, PA, USA, October 18-21, 2014*, pages 424–433, 2014. doi: 10.1109/FOCS.2014.52. URL `https://doi.org/10.1109/FOCS.2014.52`.

[66] Frank Thomson Leighton and Satish Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *J. ACM*, 46(6):787–832, 1999. doi: 10.1145/331524.331526. URL `https://doi.org/10.1145/331524.331526`.

[67] Jason Li. Faster parallel algorithm for approximate shortest path. In *ACM Symposium on the Theory of Computing (STOC)*, 5 2020. doi: 10.1145/3357713.3384268. URL `https://doi.org/10.1145/3357713.3384268`.

[68] Jason Li. Faster parallel algorithm for approximate shortest path. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 308–321, 2020.

[69] Jason Li. Deterministic mincut in almost-linear time. *STOC*, 2021.

[70] Jason Li and Debmalya Panigrahi. Deterministic min-cut in poly-logarithmic max-flows. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020*. IEEE Computer Society, 2020.

[71] Jason Li and Debmalya Panigrahi. Approximate Gomory-Hu tree is faster than $n-1$ max-flows. In *Proceedings of the 53rd Annual ACM Symposium on Theory of Computing*, 2021.

[72] Jason Li and Thatchaphol Saranurak. Deterministic weighted expander decomposition in almost-linear time, 2021.

[73] Jason Li, Danupon Nanongkai, Debmalya Panigrahi, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. Vertex connectivity in poly-logarithmic max-flows. *STOC*, 2021.

[74] Nathan Linial, Eran London, and Yuri Rabinovich. The geometry of graphs and some

of its algorithmic applications. *Combinatorica*, 15(2):215–245, 1995.

[75] Yang P Liu and Aaron Sidford. Faster divergence maximization for faster maximum flow. *arXiv preprint arXiv:2003.08929*, 2020.

[76] Yang P. Liu, Sushant Sachdeva, and Zejun Yu. Short cycles via low-diameter decompositions. In *SODA*, pages 2602–2615. SIAM, 2019.

[77] Aleksander Madry. Faster approximation schemes for fractional multicommodity flow problems via dynamic graph algorithms. In *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*, pages 121–130. IEEE Computer Society, 2010. doi: 10.1145/1806689.1806708. URL `https://doi.org/10.1145/1806689.1806708`.

[78] Aleksander Madry. Fast approximation algorithms for cut-based problems in undirected graphs. In *FOCS*, pages 245–254. IEEE Computer Society, 2010.

[79] G. A. Margulis. Explicit construction of concentrators. *Problemy Peredafi Iqfiwmacii*, 9(4):71–80, 1973. (English translation in Problems Inform. Transmission (1975)).

[80] David W Matula. A linear time 2+ ε approximation algorithm for edge connectivity. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 500–504, 1993.

[81] N. Megiddo, Arie Tamir, Eitan Zemel, and Ramaswamy Chandrasekaran. An $o(n \log^2 n)$ algorithm for the k th longest path in a tree with applications to location problems. *SIAM Journal on Computing*, 10, 05 1981. doi: 10.1137/0210023.

[82] Gary L Miller, Richard Peng, Adrian Vladu, and Shen Chen Xu. Improved parallel algorithms for spanners and hopsets. *arXiv preprint arXiv:1309.3545*, 2013.

[83] Gary L Miller, Richard Peng, and Shen Chen Xu. Parallel graph decompositions using random shifts. In *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*, pages 196–203. ACM, 2013.

[84] Sagnik Mukhopadhyay and Danupon Nanongkai. A note on isolating cut lemma for submodular function minimization. *arXiv preprint arXiv:2103.15724*, 2021.

[85] Hiroshi Nagamochi and Toshihide Ibaraki. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica*, 7(5&6):583–596, 1992. doi: 10.1007/BF01758778. URL `https://doi.org/10.1007/BF01758778`.

[86] Hiroshi Nagamochi and Toshihide Ibaraki. Computing edge-connectivity in multigraphs and capacitated graphs. *SIAM Journal on Discrete Mathematics*, 5(1):54–66, 1992.

[87] Danupon Nanongkai and Thatchaphol Saranurak. Dynamic spanning forest with worst-case update time: adaptive, Las Vegas, and $O(n^{1/2-\epsilon})$-time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 1122–1129. ACM, 2017. doi: 10.1145/3055399.

3055447. URL https://doi.org/10.1145/3055399.3055447.

[88] Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. Dynamic minimum spanning forest with subpolynomial worst-case update time. In *FOCS*, pages 950–961. IEEE Computer Society, 2017.

[89] Lorenzo Orecchia and Zeyuan Allen Zhu. Flow-based algorithms for local graph clustering. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pages 1267–1286. SIAM, 2014. doi: 10.1137/1.9781611973402.94. URL https://doi.org/10.1137/1.9781611973402.94.

[90] Richard Peng. Approximate undirected maximum flows in o (m polylog (n)) time. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 1862–1867. SIAM, 2016.

[91] Harald Räcke, Chintan Shah, and Hanjo Täubig. Computing cut-based hierarchical decompositions in almost linear time. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pages 227–238, 2014. doi: 10.1137/1.9781611973402.17. URL https://doi.org/10.1137/1.9781611973402.17.

[92] Yousef Saad and Henk A Van Der Vorst. Iterative solution of linear systems in the 20th century. *Numerical Analysis: Historical Developments in the 20th Century*, pages 175–207, 2001.

[93] Thatchaphol Saranurak. A simple deterministic algorithm for edge connectivity. In *Symposium on Simplicity in Algorithms (SOSA)*, pages 80–85. SIAM, 2021.

[94] Thatchaphol Saranurak and Di Wang. Expander decomposition and pruning: Faster, stronger, and simpler. In *SODA*, pages 2616–2635. SIAM, 2019. To appear in SODA'19.

[95] Jonah Sherman. Breaking the multicommodity flow barrier for $O(\sqrt{\log n})$-approximations to sparsest cut. In *50th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2009, October 25-27, 2009, Atlanta, Georgia, USA*, pages 363–372, 2009.

[96] Jonah Sherman. Nearly maximum flows in nearly linear time. In *FOCS*, pages 263–269. IEEE, IEEE Computer Society, 2013.

[97] Jonah Sherman. Area-convexity, l_∞ regularization, and undirected multicommodity flow. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 452–460. SIAM, 2017. doi: 10.1145/3055399.3055501. URL https://doi.org/10.1145/3055399.3055501.

[98] Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983. doi: 10.1016/0022-0000(83)90006-5. URL https://doi.org/10.1016/0022-0000(83)90006-5.

[99] D. Spielman and N. Srivastava. Graph sparsification by effective resistances. *SIAM Journal on Computing*, 40(6):1913–1926, 2011.

[100] Daniel A. Spielman and Shang-Hua Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *STOC*, pages 81–90. ACM, 2004. doi: 10.1145/1007352.1007372. URL `http://doi.acm.org/10.1145/1007352.1007372`.

[101] Daniel A. Spielman and Shang-Hua Teng. Spectral sparsification of graphs. *SIAM J. Comput.*, 40(4):981–1025, 2011. doi: 10.1137/08074489X. URL `https://doi.org/10.1137/08074489X`.

[102] Mechthild Stoer and Frank Wagner. A simple min-cut algorithm. *Journal of the ACM (JACM)*, 44(4):585–591, 1997.

[103] Jan van den Brand, Yin Tat Lee, Yang P. Liu, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. Minimum cost flows, mdps, and ℓ_1-regression in nearly linear time for dense instances. *CoRR*, abs/2101.05719, 2021. URL `https://arxiv.org/abs/2101.05719`.

[104] Christian Wulff-Nilsen. Fully-dynamic minimum spanning forest with improved worst-case update time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 1130–1143. ACM, 2017. doi: 10.1145/3055399.3055415. URL `https://doi.org/10.1145/3055399.3055415`.

[105] Neal E. Young. Randomized rounding without solving the linear program. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '95, page 170–178, USA, 1995. ISBN 0898713498.