From Graphs to Matrices, and Back: New Techniques for Graph Algorithms

by

Aleksander Mądry

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2011

© Massachusetts Institute of Technology 2011. All rights reserved.

Author .....
Department of Electrical Engineering and Computer Science
May 20, 2011

Certified by .....
Michel X. Goemans
Professor of Applied Mathematics
Thesis Supervisor

Certified by .....
Jonathan A. Kelner
Assistant Professor of Applied Mathematics
Thesis Supervisor

Accepted by .....
Professor Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Students

From Graphs to Matrices, and Back: New Techniques for Graph Algorithms

by

Aleksander Mądry

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2011, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

The growing need to deal efficiently with massive computing tasks prompts us to consider the following question:

How well can we solve fundamental optimization problems if our algorithms have to run really quickly?

The motivation for the research presented in this thesis stems from addressing the above question in the context of algorithmic graph theory.

To pursue this direction, we develop a toolkit that combines a diverse set of modern algorithmic techniques, including sparsification, low-stretch spanning trees, the multiplicative-weights-update method, dynamic graph algorithms, fast Laplacian system solvers, and tools of spectral graph theory.

Using this toolkit, we obtain improved algorithms for several basic graph problems including:

- **The Maximum s-t Flow and Minimum s-t Cut Problems.** We develop a new approach to computing $(1-\epsilon)$-approximately maximum s-t flow and $(1+\varepsilon)$-approximately minimum s-t cut in undirected graphs that gives the fastest known algorithms for these tasks. These algorithms are the first ones to improve the long-standing bound of $O(n^{3/2})$ running time on sparse graphs;

- **Multicommodity Flow Problems.** We set forth a new method of speeding up the existing approximation algorithms for multicommodity flow problems, and use it to obtain the fastest-known $(1-\epsilon)$-approximation algorithms for these problems. These results improve upon the best previously known bounds by a factor of roughly $\Omega(m/n)$, and make the resulting running times essentially match the $\Omega(mn)$ "flow-decomposition barrier" that is a natural obstacle to all the existing approaches;

- **Undirected (Multi-)Cut-Based Minimization Problems**. We develop a general framework for designing fast approximation algorithms for (multi-)cut-based minimization problems in undirected graphs. Applying this framework leads to the first algorithms for several fundamental graph partitioning primitives, such as the (generalized) sparsest cut problem and the balanced separator problem, that run in close to linear time while still providing polylogarithmic approximation guarantees;

- **The Asymmetric Traveling Salesman Problem**. We design an $O(\frac{\log n}{\log \log n})$-approximation algorithm for the classical problem of combinatorial optimization: the asymmetric traveling salesman problem. This is the first asymptotic improvement over the long-standing approximation barrier of $\Theta(\log n)$ for this problem;

- **Random Spanning Tree Generation**. We improve the bound on the time needed to generate an uniform random spanning tree of an undirected graph.

Thesis Supervisor: Michel X. Goemans
Title: Professor of Applied Mathematics

Thesis Supervisor: Jonathan A. Kelner
Title: Assistant Professor of Applied Mathematics

Acknowledgments

First and foremost, I would like to thank Michel Goemans and Jonathan Kelner for the guidance, support, and nurturing that they provided me with during my graduate studies. Michel taught me how to choose problems to work on and served as a role model of an upstanding researcher. Without Jon's mentoring, contagious enthusiasm, and endless support that was going beyond the call of duty, this thesis would not have been possible. Having both of them as my co-advisors was a truly unique experience that I will always remember. I learned a great deal from it.

I also want to thank Andrzej Borowiec, Maciej Liśkiewicz, Krzysztof Loryś, and Leszek Pacholski for their help, advice, and support at the beginning of my scientific career. If not for them, I would not be here.

I want to express my gratitude to Costis Daskalakis and Piotr Indyk for serving on my thesis committee and their frequent advice.

I am grateful to Daniel Spielman and Shang-hua Teng for their advice, as well as, for their beautiful work without which many of the results in this thesis would have never been conceived.

The MIT's theory group provided a wonderful environment for being a graduate student. I benefited from it greatly. I want to thank Costis Daskalakis, Piotr Indyk, Ronitt Rubinfeld, and Madhu Sudan, for making me feel welcome here. (Also, thanks for all the outings, Madhu!) Special thanks to all my MIT friends: Alexandr Andoni, Arnab Bhattacharyya, Erik and Martin Demaine, Andy Drucker, Bernhard Haeupler, Ankur Moitra, Jelani Nelson, Krzysztof Onak, Rotem Oshman, Debmalya Panigrahi, and Mihai Pătraşcu for making each day at the office an enjoyable experience. I am especially grateful to Alex for being my "older brother" during my first years at MIT.

I also want to thank Umesh Vazirani for hosting me during my visits to Berkeley, and Nikhil Bansal for the invitations to IBM Research. These experiences influenced me greatly and helped me develop as a researcher.

Finally, I am immensely grateful to my wife Kamila and my family (that is soon to be joined by its newest member!) for an unbelievable amount of support, love, and happiness that they have given me, as well as, for their tolerance of me during the frustrating times of my graduate studies. This thesis is dedicated to them.

To my wife Kamila,
and my family

Contents

1 Introduction **15**

1.1 Results and Structure of This Thesis 17

2 Background **21**

2.1 Preliminary Definitions 21

2.2 Cuts and Flows 21

2.3 Laplacian of a Graph 23

2.4 Electrical Flows and Resistor Networks 25

2.4.1 Effective s-t Resistance and Effective s-t Conductance 27

2.5 Electrical Flows, Random Walks, Random Spanning Trees, and Laplacians 28

2.5.1 Spectral Connection 28

2.5.2 Electrical Flows, Random Walks, and Random Spanning Trees 29

2.6 Approximating Graphs via Simpler Graphs 30

2.7 Laplacian System Solvers 33

2.8 Packing Flows via Multiplicative-Weights-Update Method 35

2.8.1 Designing a γ-approximate Solver 37

2.8.2 Correctness of the Multiplicative-Weights-Update Routine . . 39

2.8.3 Convergence of the Multiplicative-Weights-Update Routine . . 41

I Cuts and Flows **43**

3 Approximation of Maximum Flow in Undirected Graphs **45**

3.1 Introduction 45
3.1.1 Previous Work on Maximum Flows and Minimum Cuts 46
3.1.2 Outline of This Chapter 47
3.2 Preliminaries and Notations 47
3.3 A Simple $\tilde{O}(m^{3/2}\varepsilon^{-5/2})$-Time Flow Algorithm 48
3.3.1 From Electrical Flows to Maximum Flows: Oracle Construction 49
3.3.2 Computing Electrical Flows 51
3.4 An $\widetilde{O}(mn^{1/3}\varepsilon^{-11/3})$ Algorithm for Approximate Maximum Flow . . . 55
3.4.1 The Improved Algorithm 56
3.4.2 Analysis of the New Algorithm 57
3.4.3 The Proof of Lemma 3.4.1 58
3.4.4 Improving the Running Time to $\widetilde{O}(mn^{1/3}\varepsilon^{-11/3})$ 63
3.4.5 Approximating the Value of the Maximum s-t Flow in Time $\widetilde{O}(m + n^{4/3}\varepsilon^{-8/3})$ 63
3.5 A Dual Algorithm for Finding an Approximately Minimum s-t Cut in Time $\widetilde{O}(m + n^{4/3}\varepsilon^{-8/3})$ 64
3.5.1 An Overview of the Analysis 64
3.5.2 Cuts, Electrical Potentials, and Effective Resistance 65
3.5.3 The Proof that the Dual Algorithm Finds an Approximately Minimum Cut 67
3.6 Towards a Maximum Flow Algorithm for Directed Graphs 72

4 Multicommodity Flow Problems **75**
4.1 Introduction 75
4.1.1 Previous Work 76
4.1.2 Overview of Our Results 77
4.1.3 Notations and Definitions 79
4.1.4 Outline of This Chapter 80
4.2 Multiplicative-Weights-Update Routine for Solving Maximum Multicommodity Flow Problem 80

4.3 Solving Multicommodity Flow Problems and Dynamic Graph Algorithms 83
4.3.1 $(\delta, M_{\max}, M_{\min}, \mathcal{Q})$-ADSP data structure 86
4.3.2 Solving the Decremental Dynamic All-Pairs Shortest Paths Problem Using the $(\delta, M_{\max}, M_{\min}, \widehat{\mathcal{P}})$-ADSP Data Structure 87
4.4 Maximum Multicommodity Flow Problem 88
4.4.1 Existence of Short Paths in $\widehat{\mathcal{P}}$ 89
4.4.2 Randomized Cycling Through Commodities 89
4.4.3 Our Algorithm 90
4.5 Construction of the $(\delta, M_{\max}, M_{\min}, \widehat{\mathcal{P}})$-ADSP Data Structure 94
4.5.1 Implementation of the $(\delta, M_{\max}, M_{\min}, \widehat{\mathcal{P}})$-ADSP with Linear Dependence on $\frac{M_{\max}}{M_{\min}}$ 95
4.5.2 Proof of Theorem 4.3.4 99
4.6 Maximum Concurrent Flow Problem 101
4.6.1 Our Algorithm 104

5 (Multi-)Cut-Based Minimization Problems in Undirected Graphs 107
5.1 Introduction 107
5.1.1 Previous Work on Graph Partitioning 109
5.1.2 Fast Approximation Algorithms for Graph Partitioning Problems 110
5.1.3 Overview of Our Results 111
5.1.4 Overview of the Techniques 113
5.1.5 Outline of This Chapter 114
5.2 Preliminaries 114
5.2.1 The Maximum Concurrent Flow Problem 114
5.2.2 Embedability 115
5.3 Graph Decompositions and Fast Approximation Algorithms for Cut-based Problems 116
5.3.1 Finding a Good $(\alpha, \mathcal{G})$-decomposition Efficiently 118
5.4 Applications 121
5.4.1 Computing Maximum Concurrent Flow Rate on Trees 122

5.4.2 Generalized Sparsest Cut Problem 123
5.4.3 Balanced Separator and Sparsest Cut Problem 124
5.5 Proof of Theorem 5.3.6 128
5.5.1 Graphs $H(T,F)$ and the Family $\mathcal{H}[j]$ 129
5.5.2 Obtaining an $(\widetilde{O}(\log n), \mathcal{H}[\widetilde{O}(\frac{m \log U}{t})])$-decomposition of G . . 130
5.5.3 Obtaining an $(\widetilde{O}(\log n), \mathcal{G}_V[\widetilde{O}(\frac{m \log U}{t})])$-decomposition of G . . 134
5.6 Proof of Theorem 5.3.7 138

II Beyond Cuts and Flows **142**

6 Approximation of the Asymmetric Traveling Salesman Problem **143**
6.1 Introduction 143
6.2 Notation 145
6.3 The Held-Karp Relaxation 146
6.4 Random Spanning Tree Sampling and Concentration Bounds 147
6.4.1 Sampling a $\boldsymbol{\lambda}$-Random Tree 149
6.4.2 Negative Correlation and a Concentration Bound 150
6.5 The Thinness Property 151
6.6 Transforming a Thin Spanning Tree into an Eulerian Walk 153
6.7 A Combinatorial Algorithm for Finding $\widetilde{\boldsymbol{\lambda}}$s 155

7 Generation of an Uniformly Random Spanning Tree **161**
7.1 Introduction 161
7.1.1 Random Spanning Trees and Arborescences 162
7.1.2 An Outline of Our Approach 163
7.1.3 Outline of This Chapter 164
7.2 The Structure of the Walk X 164
7.2.1 (ϕ, γ)-decompositions 164
7.2.2 The Walk X 165
7.3 Our Algorithm 167
7.3.1 Simulation of the Shortcutting 168

7.3.2 Obtaining a Good (ϕ, γ)-decompositions Quickly 171
7.3.3 Generating a Random Arborescence in Time $\widetilde{O}(m^{3/2})$ 171
7.4 Obtaining an $\widetilde{O}(m\sqrt{n})$ Running Time 172
7.4.1 Strong (ϕ, γ)-decompositions 172
7.4.2 Computating of $Q_v(u)$ 174
7.4.3 Coping with Shortcomings of $\widehat{X}$ 175
7.4.4 Proof of Theorem 7.1.2 175

8 Conclusions **177**

Bibliography **181**

Chapter 1

Introduction

Optimization is one of the most basic tasks in modern science. An amazingly vast array of tasks arising in computer science, operations research, economics, biology, chemistry, and physics can be cast as optimization problems. As a result, the goal of understanding the nature of optimization from an algorithmic point of view is a cornerstone of theoretical computer science. In particular, it motivated the notion of polynomial-time algorithms as a standard of efficiency.

Unfortunately, many optimization problems of practical interest are NP-hard and thus probably cannot be solved exactly in polynomial time. One way of coping with this intractability is to change our paradigm by insisting that our algorithms produce solutions in polynomial time, and then trying to obtain a solution whose quality is within some *approximation ratio* of the optimum. Over the last thirty-five years, this approach has been extremely successful, giving rise to the field of approximation algorithms and leading to practical algorithms for a plethora of real-world optimization problems.

However, there has been an emergence in recent years of various extremely large graphs, which has changed the nature of the optimization tasks that we are facing. If one considers, for example, the graphs that companies like Google or Facebook have to deal with, one quickly realizes that the size of the corresponding optimization problems that one needs to solve on a day-to-day (or even hour-to-hour) basis is much larger than what one would have encountered just a decade ago. It becomes apparent that this explosion in the amount of data to be processed cannot be accommodated by the development of hardware-based solutions alone, especially given the growing need to perform computations in a cost- and energy-efficient manner. It is therefore critical to address this issue from an algorithmic point of view as well. As a parallel to the method employed in the past to cope with NP-hardness, one should focus in this context on reducing the running time of the algorithms to make them as fast as possible – ideally, to run in *nearly-linear* time – even if it comes at a cost of reducing the quality of the returned solution.

The research presented in this thesis stems from pursuing of the above program in the context of *graph problems*. At its core is the general question:

Can we make algorithmic graph theory run in nearly-linear time?

That is, can we obtain algorithms for graph problems that run in essentially the best possible time and that compute solutions whose quality is comparable to the best achievable by polynomial-time algorithms?

Addressing this challenge seems to be beyond the reach of classical, purely combinatorial methods. Therefore, we approach it by developing and applying a diverse set of modern algorithmic techniques, including sparsification, low-stretch spanning trees, the multiplicative-weights-update method, dynamic graph algorithms, fast Laplacian system solvers, and tools of spectral graph theory. Using these, we obtain fast approximation algorithms for the single- and multi-commodity flow problems, and a variety of cut and multi-cut problems such as the minimum s-t cut problem, the (generalized) sparsest cut problem, and the balanced separator problem.

A recurring theme that we want to highlight here is that most of the results presented in this thesis are established by connecting combinatorial properties of graphs to linear-algebraic properties of associated matrices, known as Laplacians.

It is well-known that one can use the eigenvalue spectrum of the Laplacian matrix to study the underlying graph. For example, one can use this to estimate the mixing time of the random walks or approximate certain cut properties of the graph. (An in-depth treatment of such methods can be found in the book by Chung [42].)

In this thesis, however, we show that much more can be obtained by broadening our investigation to include more general linear-algebraic properties of Laplacians. In particular, a key ingredient in many of our results is the use of *electrical flows* and related objects to probe the combinatorial structure of the graph. It turns out that this approach allows one to obtain much richer information about the graph than that conveyed by just looking at the spectrum of the Laplacian. Furthermore, computing electrical flows corresponds to solving a linear system in the Laplacian matrix (i.e., Laplacian system). One can approximately solve such a system in nearly-linear time [129, 96], thus this gives rise to an extremely efficient primitive that allows us to access the information conveyed by electrical flows in an amount of time comparable to that required by Dijkstra's algorithm.

The above approach to studying graphs provides us with a powerful new toolkit for algorithmic graph theory. In fact, as we will see in this thesis, its applicability goes beyond providing faster algorithms for graph problems. By combining the classical polyhedral methods with the notion of random spanning trees and its understanding via electrical flows, we are able to break the long-standing approximation barrier of $\Theta(\log n)$ for one of the fundamental problems of combinatorial optimization: the asymmetric traveling salesman problem.

Finally, we want to note that even though the above-mentioned bridge between the combinatorial and linear-algebraic worlds is usually used to obtain better understanding of combinatorial problems, it is also possible to utilize this connection in the other direction. Namely, in this thesis, we consider a classical algorithmic problem of

spectral graph theory: the generation of an uniformly random spanning tree in undirected graph, a problem that was traditionally approached with random-walk-based – and thus linear-algebraic in nature – methods. We show how by integrating these methods with the combinatorial graph partitioning techniques and fast Laplacian system solvers, one can speed up the existing algorithms for this problem.

1.1 Results and Structure of This Thesis

The presentation of results of this thesis is divided into two main parts. The first part focuses on finding faster algorithms for fundamental problems that deal with cuts and flows. The second one is concerned with graph problems that are outside of this envelope.

Chapter 2 – Background. We start by providing basic graph-theoretic definitions, as well as, describing some of the tools and concepts that will be used repeatedly throughout this thesis. We introduce the concepts of a Laplacian matrix of a graph and of electrical flows, and we explain the connection between them. We portray how electrical flows – and thus Laplacian matrices – are related to the behavior of random walks in the graph and to the notion of random spanning trees. Next, we introduce two important examples of techniques that allow one to approximate a given graph by a simpler one: the sparsification and low-stretch spanning trees. We explain the role of these techniques in obtaining one of the key tools employed in this thesis – nearly-linear time approximate Laplacian system solvers. Finally, we describe another technique that we will find very useful: multiplicative-weights-update method. The particular version of this method that we present is adjusted to apply to the diverse scenarios that arise in the thesis. It draws upon the variations and approaches to its analysis that were presented in [14] and [139].

Chapter 3 – The Maximum s-t Flow and Minimum s-t Cut Problems. We develop a new approach to computing $(1-\epsilon)$-approximately maximum s-t flows and $(1+\varepsilon)$-approximately minimum s-t cut in undirected graphs. Our approach employs electrical flow computations – each of them corresponds to solving a Laplacian system – to probe the global flow structure of the graph. The algorithms that result from doing this are the fastest known ones for these tasks. In particular, they improve the long-standing bound of $O(n^{3/2})$ running time on sparse graphs. The material in this chapter is based on joint work with Paul Christiano, Jonathan Kelner, Daniel Spielman, and Shang-Hua Teng [40].

Chapter 4 – Multicommodity Flow Problems. Next, we turn our attention to multicommodity flow problems, which are natural generalizations of the maximum s-t flow problem. We consider the task of designing $(1-\varepsilon)$-approximation algorithms for two of the most popular problems in this family: the maximum multicommodity flow problem and the maximum concurrent flow problem. We develop a new method of speeding up the existing algorithms for it. This method is based on employing

the ideas from an area of dynamic graph algorithms to improve the performance of the multiplicative-weights-update-based algorithms that are traditionally used in the context of multicommodity flows. As a result, we obtain running time improvements by a factor of roughly $\widetilde{O}(m/n)$ and, whenever ε is fixed, the obtained running times essentially match a natural barrier for all the existing fast algorithms and are within a $\widetilde{O}(m/n)$ factor of a trivial lower bound that applies to all the algorithms for these problems. The material in this chapter is based on [106].

Chapter 5 – (Multi-)Cut-Based Minimization Problems in Undirected Graphs. We focus on the (multi-)cut-based minimization problems – a class of problems capturing a variety of basic graph problems including the minimum cut problem, the minimum s-t cut problem, and many graph partitioning primitives such as the (generalized) sparsest cut problem and the balanced separator problem. We present a general method of designing fast approximation algorithms for this class of problems in undirected graphs. In particular, we develop a technique that given any such problem that can be approximated quickly on trees, allows us to approximate it almost as quickly on general graphs while only losing a poly-logarithmic factor in the approximation guarantee.

As an illustration of usefulness of this framework, we apply it to obtain the first algorithms for the (generalized) sparsest cut problem and the balanced separator problem that run in close to linear time while still providing polylogarithmic approximation guarantees.

One of the key ingredients in establishing this framework is exploiting the interplay between a version of the multiplicative-weights-updated-based graph decomposition technique of Räcke [115] and the ideas and tools that arise in the context of fast Laplacian system solvers [129, 96]. The material in this chapter is based on [105].

Chapter 6 – The Asymmetric Traveling Salesman Problem. We derive a randomized algorithm which delivers a solution within a factor of $O(\log n / \log \log n)$ of the optimum with high probability. This improves upon the long-standing approximation barrier of $\Theta(\log n)$. To achieve this improvement, we combine the traditional polyhedral approaches with a randomized sampling procedure that is based on employing the notion of random spanning trees and exploits their connection to electrical flows and Laplacian systems. The material in this chapter draws upon joint work with Arash Asadpour, Michel Goemans, Shayan Oveis Gharan, and Amin Saberi [18].

Chapter 7 – Generation of an Uniformly Random Spanning Tree. We give a new algorithm for a fundamental graph problem of spectral graph theory: generating uniformly random spanning trees in undirected graphs. Our algorithm produces a sample from the uniform distribution over all the spanning trees of the underlying graph in expected time $\widetilde{O}(m\sqrt{n})$. This improves the sparse graph case of the best previously known worst-case bound of $O(\min\{mn, n^{2.376}\})$.

To achieve this improvement, we exploit the connection between random walks on graphs and electrical networks to integrate the traditional random-walk-based

techniques for the problem with combinatorial graph partitioning primitives and fast Laplacian system solvers. The material in this chapter is based on joint work with Jonathan Kelner and James Propp [91].

Chapter 8 – Conclusions. We summarize what was achieved in this thesis, and discuss the possible avenues for further research.

Chapter 2

Background

In this chapter, we provide the basic graph-theoretic definitions and introduce some of the tools and concepts that will be employed throughout this thesis. This includes the concept of a Laplacian matrix of a graph and of electrical flows, as well as, tools such as Laplacian system solvers, multiplicative-weights-update method, random and low-stretch spanning trees, and sparsification.

2.1 Preliminary Definitions

The main objects of study of this thesis are graphs. They might be directed or undirected. Some of the presented results, however, apply only to undirected setting. Graphs are denoted as $G = (V, E)$, where V is the set of vertices and E is the set of edges. Whenever we want to emphasize that the edges are directed, we will call them *arcs*. Usually, the variables n and m will denote the number of vertices $|V|$ and, respectively, edges $|E|$ of the graph being considered. Also, each edge (arc) e of our graphs might sometimes be equipped with additional characteristics: *capacity* u_e, *length* l_e and/or *weight* w_e. These quantities are always assumed to be non-negative.

We will often use the "soft-O" notation, i.e., we define $\widetilde{O}(g)$ to denote a quantity that is $O(g \log^c g)$, for some constant c.

2.2 Cuts and Flows

The notions that we will be dealing with extensively are the notions of a cut, an s-t cut, and an s-t flow. We define a *cut* to be any subset of vertices C that induces a non-trivial partition of vertices, i.e., any C that is non-empty and is not equal to the set of all vertices V. For undirected graphs, we denote by $E(C)$ the set of edges with exactly one endpoint in C. For graphs that are directed, $E(C)$ denotes the set of edges that *leave* the cut C, i.e., the set of edges with their tails being in C and their heads being in the complement $\overline{C}$ of the set C. Given some capacities $\{u_e\}_e$ corresponding to edges, we define $u(C)$ to be the *capacity of the cut* C and make it equal to the total capacity of the edges in $E(C)$, i.e., $u(C) := \sum_{e \in E(C)} u_e$.

By an s-t *cut*, for given vertices s and t, we mean any cut C in G that separates s from t, i.e., such that $s \in C$, but $t \notin C$.

Now, for a directed graph $G = (V, E)$ and two of its vertices s and t, we define an s-t flow to be a function that assigns non-negative values to arcs and obeys the *flow-conservation constraints*:

$$\sum_{e \in E^-(v)} f(e) - \sum_{e \in E^+(v)} f(e) = 0 \quad \text{for all } v \in V \setminus \{s, t\},$$

where $E^-(v)$ (resp. $E^+(v)$) is the set of arcs with v being their tail (resp. head). The *value* $|f|$ of the flow is defined to be the net flow out of the source vertex,

$$|f| := \sum_{e \in E^-(s)} f(e) - \sum_{e \in E^+(s)} f(e).$$

It follows easily from the flow conservation constraints that the net flow out of s is equal to the net flow into t, so $|f|$ may be interpreted as the amount of flow that is sent from the *source* s to the *sink* t.

To extend the above definition to undirected graphs, one could just model undirected edges as two arcs that are oriented in opposite directions. However, for notational convenience, we will make our definition of s-t flow in undirected graph slightly different. Namely, given an undirected graph $G = (V, E)$, we orient each edge in E arbitrarily; this divides the edges incident to a vertex $v \in V$ into the set $E^+(v)$ of edges oriented towards v and the set $E^-(v)$ of edges oriented away from v. These orientations will be used to interpret the meaning of a positive flow on an edge. If an edge has positive flow and is in $E^+(v)$, then the flow is towards v. Conversely, if it has negative flow then the flow is away from v. One should keep in mind that we allow here the flow on an edge to go in either direction, regardless of this edge's orientation.

Now, once the sets $E^+(v)$ and $E^-(v)$ are introduced, the rest of the definition of an s-t flow is analogous to the one in the directed case. The only difference is that now f can also assign negative values to edges (as they correspond to flowing in the direction opposite to the orientation).

Cut-based graph problems: There is a lot of optimization graph problems that are related to cuts. Usually, they correspond to a task of finding a cut that partitions the graph into two parts while minimizing the capacity of edges cut and possibly satisfying some additional constraints. The most basic cut-based problem is the *minimum cut problem* in which our goal is to simply find a cut in G of minimum capacity. Another fundamental cut-based graph problems called *minimum s-t cut problem* in which we are given two vertices: source s and sink t, and we want to find a minimum capacity s-t cut of the graph, i.e., one that separates the source s from the sink t.

Sometime, however, instead of just minimizing the capacity of the edges cut, we are interested in also making sure that both sides of the cut are relatively large. To

achieve this, one can focus on minimizing the ratio of the capacity of the cut to some measure of the size of its smaller side. For example, in the *minimum conductance cut problem* we are given a capacitated graph $G = (V, E, \boldsymbol{u})$ and we want to find a cut C that minimizes its *conductance*:

$$\frac{u(C)}{\min\{Vol(C), Vol(\overline{C})\}}, \tag{2.1}$$

where $Vol(C)$ (resp. $Vol(\overline{C})$) is the sum of capacities of all the edges with at least one endpoint in C (resp. $\overline{C}$). Another important problem of this type is the *sparsest cut problem* in which we want to find a cut C with minimal *sparsity*:

$$\frac{u(C)}{\min\{|C|, |\overline{C}|\}}. \tag{2.2}$$

2.3 Laplacian of a Graph

A matrix that will play a fundamental role in this thesis is the Laplacian of the graph. As we will see, this matrix has intimate connections to a variety of graph-theoretic notions. In particular, one can relate linear-algebraic properties of the Laplacian to the behavior of random walks, the sampling probabilities of random spanning trees, as well as, the combinatorial cut-flow structure of the underlying graph.

To define the Laplacian matrix, let us consider a weighted *undirected*[1] graph $G = (V, E, \boldsymbol{w})$. Recall that the *adjacency matrix* $\boldsymbol{A}(G)$ of this graph is given by

$$A(G)_{u,v} := \begin{cases} w_e & \text{if the edge } e = (u, v) \text{ is in } E, \\ 0 & \text{otherwise.} \end{cases}$$

Now, the Laplacian $\boldsymbol{L}(G)$ of the graph G is defined as

$$\boldsymbol{L}(G) := \boldsymbol{D}(G) - \boldsymbol{A}(G),$$

where $\boldsymbol{D}(G)$ is a diagonal matrix with $D(G)_{vv}$ equal to the weighted degree $\sum_{e \in E(v)} w_e$ of the vertex v. (Here, we denoted by $E(v)$ the set $E(\{v\})$ of edges adjacent to v.)

One can easily check that the entries of the Laplacian are given by

$$L(G)_{u,v} := \begin{cases} \sum_{e \in E(v)} w_e & \text{if } u = v, \\ -w_e & \text{if the edge } e = (u, v) \text{ is in } E, \\ 0 & \text{otherwise.} \end{cases}$$

An important property of the Laplacian – one that reveals a lot of its structure – is that it can be expressed as a product of very simple matrices. Namely, if we

[1] One could also extend the definition of the Laplacian that we are about to present to the case of directed graphs. However, the resulting object loses most of the properties that are useful to us. Therefore, we consider the Laplacian only in the undirected setting.

consider the *edge-vertex incidence matrix* $\boldsymbol{B}$, which is an $n \times m$ matrix with rows indexed by vertices and columns indexed by edges, such that

$$\boldsymbol{B}_{v,e} = \begin{cases} 1 & \text{if } e \in E^-(v), \\ -1 & \text{if } e \in E^+(v), \\ 0 & \text{otherwise.} \end{cases}$$

Then, we can express the Laplacian as

$$\boldsymbol{L}(G) = \boldsymbol{B}\,\boldsymbol{W}\,\boldsymbol{B}^T, \tag{2.3}$$

where $\boldsymbol{W}$ is an $m \times m$ diagonal matrix with $W_{e,e} := w_e$.

Now, from the above factorization, one can immediately see that the quadratic form corresponding to the Laplacian can be expressed very cleanly as

$$\boldsymbol{x}^T \boldsymbol{L}(G)\boldsymbol{x} = \sum_{e=(u,v)\in E} w_e(x_u - x_v)^2. \tag{2.4}$$

This quadratic form reveals a lot of basic properties of the Laplacian. For instance, from it one can immediately see that if $\lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$ are the eigenvalues of the Laplacian $\boldsymbol{L}(G)$ and $\boldsymbol{v}_1, \boldsymbol{v}_2 \ldots, \boldsymbol{v}_n$ are the corresponding eigenvectors then:

1. All λ_is are non-negative, i.e., the Laplacian $\boldsymbol{L}(G)$ is a *positive semi-definite* matrix;
2. λ_1 is equal to 0 and the corresponding eigenvector $\boldsymbol{v}_1$ is a normalized all-ones vector $\mathbf{1}$;
3. If G is connected (which will be always the case in this thesis) λ_2 is strictly positive.

Also, the quadratic form (2.4) highlights the crucial connection between Laplacian and the cut structure of the underlying graph. Namely, if we consider a cut C in G and its *characteristic vector* $\boldsymbol{\chi}_C \in \{-1, 1\}^{|V|}$ that has coordinates corresponding to vertices in C equal to 1, and the remaining coordinates equal to -1, then we will have that

$$\boldsymbol{\chi}_C^T \boldsymbol{L}(G)\boldsymbol{\chi}_C = 4w_G(C), \tag{2.5}$$

where $w_G(C)$is the total weight in G of edges in the cut C. That is, the quadratic form of the Laplacian on such characteristic vectors measures the capacity of the corresponding cuts.

Finally, we should observe that the Laplacian matrix is not invertible (as λ_1 is equal to 0). However, it will be useful to employ a notion that allows us to invert a Laplacian in a restricted sense. This notion is the *Moore-Penrose pseudoinverse* $\boldsymbol{L}(G)^\dagger$ of $\boldsymbol{L}(G)$ that is defined as

$$\boldsymbol{L}(G)^\dagger := \sum_{i:\lambda_i \neq 0} \frac{1}{\lambda_i} \boldsymbol{v}_i \boldsymbol{v}_i^T. \tag{2.6}$$

As one can see, $\boldsymbol{L}(G)^\dagger$ acts as an inverse on the range of the matrix $\boldsymbol{L}(G)$ and as a zero matrix on the orthogonal complement of this range.

2.4 Electrical Flows and Resistor Networks

The second, after Laplacians, key concept that we need to introduce, is the notion of electrical flows. This notion arises from treating an undirected graph G as a resistor network in which each edge e has some resistance r_e; and studying the currents that are induced in this network upon applying a voltage difference to some of its vertices. As we will see shortly, electrical flows have a very close connection to the Laplacian matrices and this connection will be pivotal for some of the results we present in this thesis.

We start by defining the notion of electrical flow formally. To this end, consider a vector $\boldsymbol{r}$ of resistances of edges and an s-t flow f. Let us define the *energy* $\mathcal{E}_{\boldsymbol{r}}(f)$ of this flow f (with respect to $\boldsymbol{r}$) as

$$\mathcal{E}_{\boldsymbol{r}}(f) := \sum_e r_e f^2(e).$$

Now, the *electrical (s-t) flow of value F (with respect to $\boldsymbol{r}$)* is the (unique) s-t flow that minimizes $\mathcal{E}_{\boldsymbol{r}}(f)$ among all s-t flows f of value F.

From a physical point of view, the electrical flow of value F corresponds to the current that is induced in G if we view it as an electrical circuit in which each edge e has resistance of r_e, and we send F units of current from s to t, say by attaching s to a current source and t to ground.

It is well-known (see e.g. [104]) that an electrical flow is a *potential flow*, which means that for any electrical flow f there exists a vector $\boldsymbol{\phi} \in \mathbb{R}^n$ of *vertex potentials* that determine the flow f via *Ohm's Law*:

$$f(e) = \frac{\phi_v - \phi_u}{r_{(u,v)}} \quad \text{for each } e = (u, v). \tag{2.7}$$

As a result, finding an electrical flow boils down to finding the corresponding vertex potentials.

Now, the key connection between electrical flows and Laplacian matrices is that these vertex potentials and, thus, the electrical flow, can be obtained by solving a *Laplacian system*, i.e., a linear system of equations where the constraint matrix is the Laplacian matrix.

To establish this connection formally, let us consider a task of finding an electrical s-t flow f of value 1. (We assume that the value of the flow is 1 here, but one can get a flow of arbitrary value F by just multiplying this value-one flow by F.)

Let us view our flow f as a vector $\boldsymbol{f} \in \mathbb{R}^m$ of m real entries – one entry per edge – with the orientations of the edges determining the signs of the coordinates. It is easy to see that in this vector interpretation, the v^{th} entry of the vector $\boldsymbol{Bf}$ will be the difference between the flow out of and the flow into vertex v. Therefore, the

constraints on f that 1 unit of flow is sent from s to t and that flow is conserved at all other vertices can be compactly written as

$$\boldsymbol{B}\boldsymbol{f} = \boldsymbol{\chi}_{s,t}, \tag{2.8}$$

where $\boldsymbol{\chi}_{s,t}$ is a vector with a 1 in the coordinate corresponding to s, a -1 in the coordinate corresponding to t, and all other coordinates equal to 0.

Furthermore, by using the Ohm's Law (2.7), we can express the vector representation $\boldsymbol{f}$ of the flow f in terms of the vector of the corresponding vertex potentials $\boldsymbol{\phi}$ as

$$\boldsymbol{f} = \boldsymbol{C}\boldsymbol{B}^T\boldsymbol{\phi}, \tag{2.9}$$

where $\boldsymbol{C}$ is an $m \times m$ diagonal matrix with each diagonal entry $C_{e,e}$ being equal to the *conductance* $c_e := 1/r_e$ of the edge e.

By multiplying both sides of (2.9) by $\boldsymbol{B}$ and combining it with (2.8), we obtain that the vertex potentials $\boldsymbol{\phi}$ have to obey the following equality

$$\boldsymbol{B}\boldsymbol{C}\boldsymbol{B}^T\boldsymbol{\phi} = \boldsymbol{\chi}_{s,t}.$$

Now, by looking at the factorization of the Laplacian from (2.3) we can conclude that the vertex potentials $\boldsymbol{\phi}$ corresponding to the electrical s-t flow f of value 1 are given by a solution to the Laplacian system

$$\boldsymbol{L}(G)\boldsymbol{\phi} = \boldsymbol{\chi}_{s,t}, \tag{2.10}$$

where $\boldsymbol{L}(G)$ is the Laplacian of the underlying graph with the weight w_e equal to the conductances c_e for each e (and thus being inverses of the resistances r_e).

In other words, $\boldsymbol{\phi}$ can be obtained by multiplying the pseudo-inverse $\boldsymbol{L}(G)^\dagger$ of the corresponding Laplacian by the vector $\boldsymbol{\chi}_{s,t}$

$$\boldsymbol{\phi} = \boldsymbol{L}(G)^\dagger\boldsymbol{\chi}_{s,t}.$$

This, in turn, allows us to use the Ohm's Law (2.7) again to conclude that the vector representation $\boldsymbol{f}$ of our electrical flow f can be written compactly as

$$\boldsymbol{f} = \boldsymbol{C}\boldsymbol{B}^T\boldsymbol{L}(G)^\dagger\boldsymbol{\chi}_{s,t}. \tag{2.11}$$

At this point we also want to note that the above discussion allows us to express the energy $\mathcal{E}_r(f)$ of an electrical s-t flow f (of arbitrary value) as

$$\begin{aligned} \mathcal{E}_r(f) = \boldsymbol{f}^T\boldsymbol{R}\boldsymbol{f} &= \left(\boldsymbol{\chi}_{s,t}{}^T\boldsymbol{L}(G)^{\dagger T}\boldsymbol{B}\boldsymbol{C}^T\right)\boldsymbol{R}\left(\boldsymbol{C}\boldsymbol{B}^T\boldsymbol{L}(G)^\dagger\boldsymbol{\chi}_{s,t}\right) \\ &= \boldsymbol{\chi}_{s,t}\boldsymbol{L}(G)^\dagger\boldsymbol{L}(G)\boldsymbol{L}(G)^\dagger\boldsymbol{\chi}_{s,t} = \boldsymbol{\chi}_{s,t}{}^T\boldsymbol{L}(G)^\dagger\boldsymbol{\chi}_{s,t} = \boldsymbol{\phi}^T\boldsymbol{L}(G)\boldsymbol{\phi}, \end{aligned} \tag{2.12}$$

where $\boldsymbol{R} = \boldsymbol{C}^{-1}$ is the diagonal matrix with $R_{e,e} = r_e = 1/c_e$.

2.4.1 Effective s-t Resistance and Effective s-t Conductance

Apart of the definition of the electrical flows, there are two other basic quantities from the theory of electrical networks that we will find useful: the effective s-t resistance and effective s-t conductance.

Let f be the electrical s-t flow of value 1, and let $\boldsymbol{\phi}$ be its vector of vertex potentials. The *effective s-t resistance of G with respect to the resistances $\boldsymbol{r}$* is given by

$$R^{s,t}_{\text{eff}}(\boldsymbol{r}) := \phi_s - \phi_t. \tag{2.13}$$

(For notation convenience, we suppress the reference to s and t in our notation – and simply write $R_{\text{eff}}(\boldsymbol{r})$ – whenever these vertices are clear from the context.)

Using our linear-algebraic description of the electrical flow and equation (2.12), we have

$$R^{s,t}_{\text{eff}}(\boldsymbol{r}) = \phi_s - \phi_t = \boldsymbol{\chi_{s,t}}^T \boldsymbol{\phi} = \boldsymbol{\chi_{s,t}}^T \boldsymbol{L}(G)^{\dagger} \boldsymbol{\chi_{s,t}} = \mathcal{E}_{\boldsymbol{r}}(f).$$

This gives us an alternative description of the effective s-t resistance as the energy of the corresponding electrical flow of value 1.

Now, the related notion of the *effective s-t conductance of G with respect to the resistances $\boldsymbol{r}$* is defined as

$$C^{s,t}_{\text{eff}}(\boldsymbol{r}) = 1/R^{s,t}_{\text{eff}}(\boldsymbol{r}).$$

We note that $C^{s,t}_{\text{eff}}(\boldsymbol{r})$ is equal to the value of the electrical flow in which the difference $\phi_s - \phi_t$ of corresponding vertex potentials at s and t is 1. (Similarly to the case of effective resistance, we will suppress the reference to s and t in notation whenever they will be clear from the context.)

The following standard fact (see [32] for a proof) will be useful to us

Fact 2.4.1. *For any $G = (V, E)$ and any vector of resistances $\boldsymbol{r}$,*

$$C^{s,t}_{\text{eff}}(\boldsymbol{r}) = \min_{\substack{\boldsymbol{\phi} \mid \phi_s = 1, \\ \phi_t = 0}} \sum_{(u,v) \in E} \frac{(\phi_u - \phi_v)^2}{r_{(u,v)}}.$$

Furthermore, the equality is attained for $\boldsymbol{\phi}$ being vector of vertex potentials corresponding to the electrical s-t flow of G (with respect to $\boldsymbol{r}$) of value $1/R^{s,t}_{\text{eff}}(\boldsymbol{r})$.

This fact will allow for better understanding of how changes of the resistance of a particular edge e influence the effective resistances of other edges. In particular, we can prove the following simple, but fundamental result.

Corollary 2.4.2 (Rayleigh Monotonicity Principle)**.** *If $r'_e \geq r_e$ for all $e \in E$, then $R^{s,t}_{\text{eff}}(\boldsymbol{r}') \geq R^{s,t}_{\text{eff}}(\boldsymbol{r})$.*

Proof. For any $\boldsymbol{\phi}$,

$$\sum_{(u,v) \in E} \frac{(\phi_u - \phi_v)^2}{r'_{(u,v)}} \leq \sum_{(u,v) \in E} \frac{(\phi_u - \phi_v)^2}{r_{(u,v)}},$$

so the minimums of these expressions over possible values of ϕ obey the same relation, and thus – by Fact 2.4.1 – $C_{\text{eff}}^{s,t}(\boldsymbol{r}') \leq C_{\text{eff}}^{s,t}(\boldsymbol{r})$. Inverting both sides of this inequality yields the desired result. □

2.5 Electrical Flows, Random Walks, Random Spanning Trees, and Laplacians

As we mentioned, one reason behind fundamental importance of the Laplacian matrix is that there is a great deal of information about the graph G that one can infer from analyzing the properties of its Laplacian $\boldsymbol{L}(G)$. Here, we survey some of the examples illustrating that.

2.5.1 Spectral Connection

A particularly important example here is an approach to analyzing a graph by studying the spectrum of the associated Laplacian – most notably, its second eigenvalue λ_2. This approach, known as spectral graph theory, was started by Fiedler [64] and produced a host of interesting connections. Here, we describe only two – arguably the most well-known one – among them and for a more in-depth study we refer the reader to [126] and the book of Chung [42].

Both these connections are related to, so called, *spectral gap* $\widehat{\lambda}_2$ of G, which is just the second-smallest eigenvalue of the normalized version of the Laplacian $\boldsymbol{L}(G)$ of G. The normalized Laplacian $\boldsymbol{L}(G)$ of G is given by

$$\widehat{\boldsymbol{L}}(G) := \boldsymbol{D}(G)^{-1/2}\boldsymbol{L}(G)\boldsymbol{D}(G)^{-1/2},$$

where we recall that $\boldsymbol{D}(G)$ is the diagonal degree matrix of G.

Mixing time of random walks: The first of them is the one to the mixing time of random walks. Recall that a random walk in G corresponds to a stochastic process in which we start at some starting vertex v_0 and repeat for some number of times a step such that if we are at a vertex v we move to one of its neighbors u with probability proportional to the weight $w_{(v,u)}$ of the corresponding edge. Now, let $\tau(\varepsilon)$ be the number of steps needed for a distribution on vertices of G induced by performing a *lazy* random walk[2] started at an arbitrary vertex, to converge point-wisely to within

[2]A lazy random walk is a variation of the original random walk in which in each step we have a probability of $1/2$ of staying at the vertex we are at and, with the remaining probability of $1/2$, a step of the original random walk is made. One of the reasons why we consider this modified version of the random walk is that it will always have a unique stationary distribution, no matter what the underlying graph G is. This would not be the case for the original random walk if the graph G was, e.g., bipartite.

ε of the stationary distribution. It turns out that

$$\tau(\varepsilon) \leq \frac{1}{\widehat{\lambda}_2 \log(n/W\varepsilon)}, \tag{2.14}$$

where W is the ratio of the largest to smallest edge-weight in G.

As one can construct examples of graph for which this upper bound is essentially tight, this allows us to conclude that the mixing time of random walks is essentially governed by the value of $\widehat{\lambda}_2$. Motivated by this connection, expander graphs (that are known for having their mixing time very small) are often defined as graphs with their spectral gap bounded away (by some constant) from zero.

Cheeger's inequality: The second connection is the one to cut structure of the graph. Namely, it turns out that there is an intimate relation between the spectral gap $\widehat{\lambda}_2$ and the conductance Φ_G of G defined as $\Phi_G := \min_{\emptyset \neq C \subset V} \Phi(C)$, where the conductance $\Phi(C)$ of a cut C is given by expression (2.1) in Section 2.2. This connection is inspired by, so called, Cheeger's inequality [38] that was discovered in the context of Riemannian manifolds, and a discrete version of it (see [99, 9, 6, 55, 136, 125]) asserts that

$$2\Phi_G \geq \widehat{\lambda}_2 \geq \Phi_G^2/2.$$

Note that as cuts with small conductance are obvious obstacles to mixing of a random walk, this inequality tells us that in fact such sets are the only obstacle. To observe why it is the case, note that if there is no cuts with small conductance then the above inequality implies that the spectral gap is large and thus, by inequality (2.14), we know that the mixing time of the random walk is fast.

2.5.2 Electrical Flows, Random Walks, and Random Spanning Trees

Another type of connection between Laplacian matrices and various graph-theoretic objects is established via the notion of electrical flows, which are tightly related to Laplacians via equation (2.11).

Electrical flows and random walks: There is a lot of connections between properties of electrical flow with respect to resistances $\boldsymbol{r}$ and of the random walks in the graph with weights w_e of each edge e being equal to $1/r_e$. Here, we mention only one of them – we will need it later – but we refer the reader to the book [104] for a more complete exposition.

Fact 2.5.1 (see [104])**.** *Let $p_{s,t}(u)$ be the probability that a random walk started at vertex u will visit the vertex s before visiting the vertex t, and let $\boldsymbol{\phi}$ be the vertex potentials corresponding to the electrical s-t flow of value 1, we have:*

$$p_{s,t}(u) = \frac{\phi_u - \phi_t}{\phi_s - \phi_t}. \tag{2.15}$$

Electrical flows and random spanning trees: Yet another objects to which electrical flows (and thus Laplacians) have connection to, is the notion of random spanning trees. Given a weighted graph $G = (V, E, \boldsymbol{w})$ let us consider a stochastic procedure of choosing a spanning tree T of G (from the set of all its spanning trees) with probability proportional to $\prod_{e \in T} w_e$. The following beautiful fact ties the resulting distribution over the edges of the graph and their effective resistance – its proof can be found in Chapter 4 of [104].[3]

Fact 2.5.2 (see [104])**.** *Consider a vector of resistances $\boldsymbol{r}$ in which the resistance $r_{e'}$ of each edge e' is taken to be $1/w_{e'}$. For any edge $e = (u, v)$, the probability that this edge e is included in a random spanning tree is equal to $R^{u,v}_{\text{eff}}(\boldsymbol{r})/r_e$.*

The second fundamental connection between the random spanning trees and Laplacians is captured in the following fact.

Fact 2.5.3 (Kirchhoff's Matrix Tree Theorem [93])**.** *We have*

$$\sum_T \prod_{e \in T} w_e = \frac{1}{n} \prod_{i=2}^{n} \lambda_i.$$

Note that the right-hand side term is equal to an absolute value of any cofactor of the Laplacian. Furthermore, when all the weights w_e are equal to 1, the left-hand side term simplifies to the number of spanning trees of the underlying graph.

2.6 Approximating Graphs via Simpler Graphs

A paradigm of approximating (in various senses) the graphs by simpler ones has been very successful in design of algorithms. It often enabled one to reduce the question about the original graph to a – usually much more tractable – question about the simpler graph. Two classical examples here are spanners [10, 59, 133, 24] and cut sparsifiers of Benczúr and Karger [26]. In case of spanners, one aims for finding a sparse graph whose distances approximately preserves the distances in the original, not necessarily sparse, graph. In case of cut sparsification, one tries to find a sparse graph that approximately preservers the capacity of all the cuts of the original graph.

This paradigm of graph approximation will play a key role in this thesis. In particular, two tools that are helpful in this context and that will be relevant to the results of this thesis are: (spectral and cut) sparsification, and low-stretch spanning tree. We introduce these tools now.

[3] Lyon and Peres prove this fact only in the case of T being a uniform spanning tree i.e. when all w_es are equal, but Section 4.1 of [104] contains a justification why this proof implies this property also in the case of arbitrary w_es. Roughly speaking, for rational w_es, the main idea is to replace each edge e with Cw_e edges (for an appropriate choice of C) and consider a uniform spanning tree in the corresponding multigraph. The irrational case follows from a limit argument.

Sparsification: The goal of the (spectral) sparsification is to find for a given dense graph G, another graph $\widetilde{G}$ that is sparse and has its spectral picture approximately the same as the one of G, i.e., the quadratic form (2.4) corresponding to the Laplacian $\boldsymbol{L}(G)$ of the graph G is approximately the same as the quadratic form corresponding to the Laplacian $\boldsymbol{L}(\widetilde{G})$ of the graph $\widetilde{G}$.

It turns out that irrespectively of the density and structure of the graph G, one can always take $\widetilde{G}$ to be really sparse while still have such a spectral picture of $\widetilde{G}$ differ not too much from the original one. More precisely, inspired by the result of Benczúr and Karger [26], Spielman and Teng [129] – and later Spielman and Srivastava [127] – proved the following theorem.

Theorem 2.6.1. *Given a weighted graph $G = (V, E, \boldsymbol{w})$ and an accuracy parameter $\delta > 0$, there is a Monte Carlo algorithm that finds in $\widetilde{O}(|E|)$ time a (weighted) subgraph $\widetilde{G} = (V, \widetilde{E}, \widetilde{\boldsymbol{w}})$ of G such that:*

(1) $\widetilde{G}$ has $O(\delta^{-2}|V|\log|V|)$ edges

(2) For any vector $\boldsymbol{x}$, $\boldsymbol{x}^T\boldsymbol{L}(\widetilde{G})\boldsymbol{x} \leq \boldsymbol{x}^T\boldsymbol{L}(G)\boldsymbol{x} \leq (1+\delta)\boldsymbol{x}^T\boldsymbol{L}(\widetilde{G})\boldsymbol{x}$

The graph $\widetilde{G}$ with the above properties is called a *(spectral) sparsifier* of G.

Note that an important part of the above theorem is that we are able to construct $\widetilde{G}$ in time that is only nearly-linear in the size G. In this way, the benefit of being able to deal with the sparse graph $\widetilde{G}$ instead of dense G, is not outweighed by a prohibitively large construction time. Interestingly, if having just a polynomial-time construction of $\widetilde{G}$ is sufficient for one's purposes, one can get even better sparsifier – one that has only $O(\delta^{-2}|V|)$ instead of $O(\delta^{-2}|V|\log|V|)$ edges [25].

By employing the connections that we outlined in Section 2.5, one can show the preservation of the quadratic form of the Laplacian (cf. condition (2) in the above theorem) implies that G and $\widetilde{G}$ are similar in a variety of aspects. The particular similarity that we will be often using is that of the cut structure of G and $\widetilde{G}$. Namely, as we noted in Section 2.3 (cf. equation (2.5)), the quadratic form of the Laplacian captures, in particular, the cuts of the underlying graph. Therefore, the condition (2) in Theorem 2.6.1 can be viewed, in particular, as a statement about preservation of cuts. As a result, if we take the weights w_e to correspond to the capacities u_e of a given capacitated graph $G = (V, E, \boldsymbol{u})$, we can obtain the following corollary of Theorem 2.6.1 that was originally proved by Benczúr and Karger [26].

Corollary 2.6.2 (see [26])**.** *Given a capacitated graph $G = (V, E, \boldsymbol{u})$ and an accuracy parameter $\delta > 0$, there is a Monte Carlo algorithm that finds in $\widetilde{O}(|E|)$ time a (capacitated) subgraph $\widetilde{G} = (V, \widetilde{E}, \widetilde{\boldsymbol{u}})$ of G such that:*

(i) $\widetilde{G}$ has $O(\delta^{-2}|V|\log|V|)$ edges

(ii) For any cut $\emptyset \neq C \neq V$ we have $\widetilde{u}(C) \leq u(C) \leq (1+\delta)\widetilde{u}(C)$

To differentiate the graph $\widetilde{G}$ with the above properties from the one returned by the (more general) Theorem 2.6.1, we will call the former the *cut sparsifier* of G.

Clearly, in situations in which we are dealing with a dense graph G on which we want to solve a cut-based problem and when we can afford returning only δ-approximately optimal answer, the above corollary allows us to deal with the sparse graph $\widetilde{G}$ instead of G.

Low-stretch spanning trees: The notion of low-stretch spanning trees was introduced for the first time by Alon, Karp, Peleg and West [8] in the context of the k-server problem. Since then, however, these trees turned out to be useful in many other contexts – one of them will be presented in this thesis – including preconditioning of Laplacian matrices – see Section 2.7.

To define this notion, let us consider some spanning tree T of a weighted graph $G = (V, E, \boldsymbol{w})$. For a given edge e let us define the *stretch* $\mathsf{stretch}_T^{\boldsymbol{w}}(e)$ of e (with respect to T and $\boldsymbol{w}$) as

$$\mathsf{stretch}_T^{\boldsymbol{w}}(e) = \frac{\sum_{e' \in \mathsf{path}_T(e)} w_{e'}}{w_e},$$

where $\mathsf{path}_T(e)$ is the unique path in T that connects the endpoints of the edge e. Note that if we interpret the weights as lengths of the edges, then $\mathsf{stretch}_T^{\boldsymbol{w}}(e)$ corresponds to a ratio of the distance between the endpoints of the edge e in T to the length of the edge e.

Now, the natural question to ask in the above context is: can we always find a tree T so as all the edges e of the graph G have small stretch with respect to it? Unfortunately, if one considers G being an unweighted cycle then one immediately sees that no matter which spanning tree is chosen there always will be an edge that suffers a very high stretch of $n - 1$.

However, somewhat surprisingly, if one relaxes the above requirement and focuses on obtaining low *average* stretch then trees that yield such a small-on-average stretch always exist. In particular, the following theorem was proved by Abraham, Bartal and Neiman [1] by building on the work of Elkin, Emek, Spielman and Teng [58].[4] (Also, see [8] for a previous result, and [62] for a related work.)

Theorem 2.6.3 ([1])**.** *There is an algorithm working in $\widetilde{O}(|E|)$ time that for any weighted graph $G = (V, E, \boldsymbol{w})$ generates a spanning tree T of G such that the average stretch $\frac{1}{|E|}\sum_{e \in E} \mathsf{stretch}_T^{\boldsymbol{w}}(e)$ of edges of G is $\widetilde{O}(\log |V|)$.*

[4]Technically, the notion of stretch considered in [1] is slightly different – it corresponds to the ratio of distance of the endpoints of the edge e in T and the corresponding distance in the original graph G. But, as the latter has to be always at most w_e, the statement of the Theorem 2.6.3 is implied by [1].

2.7 Laplacian System Solvers

The task of solving Laplacian systems is one of the fundamental problems of scientific computing. It has a broad array of applications (see, e.g., [126]). From the point of view of this thesis, the key reason for our interest in them is that, as we noted in Section 2.4 (cf. equation (2.10)), the task of finding the vertex potential that describe electrical flows can be cast as the task of solving a Laplacian system. Therefore, being able to solve such systems efficiently allows us to make the connections we surveyed in Section 2.5 algorithmic.

The classical approach to solving Laplacian systems – and, more generally, linear systems – is based on, so called, direct methods. These methods can be viewed as variants of Gaussian elimination and give exact solutions. Unfortunately, due to their construction, they tend to have fairly large running time. The fastest known methods require $O(n^\omega)$ running time [35, 34], where ω is the exponent of matrix multiplication and is known to be between 2 and 2.376 [46].

As the sizes of instances we want to handle are very large, even $O(n^2)$ running time is prohibitive from our point of view. This motivates us to turning our attention to a different approach to solving Laplacian systems: the one based on iterative (indirect) methods.

These methods consist of finding successfully better approximations of the solution and tend to be much faster than the direct methods, but pay a price of providing only approximate answers. The most fundamental method in this family is the Conjugate Gradient method. This method considers a Laplacian system

$$\boldsymbol{L}(G)\boldsymbol{x} = \boldsymbol{b},$$

where $\boldsymbol{b}$ is in the range of $\boldsymbol{L}(G)$ (when the graph G is connected this means that $\boldsymbol{b}$ is orthogonal to all-ones vector $\mathbf{1}$) and, for any $\varepsilon > 0$, provides an ε-approximate solution $\widetilde{\boldsymbol{x}}$ to this system after at most $O(\sqrt{\kappa_f(\boldsymbol{L}(G))}\log 1/\varepsilon)$ iterations. Here, $\kappa_f(\boldsymbol{L}(G))$ denotes the *finite condition number* of the Laplacian $\boldsymbol{L}(G)$ that is equal to the ratio of its largest to smallest *non-zero* eigenvalue, and $\widetilde{\boldsymbol{x}}$ ε-approximates the optimal solution $\boldsymbol{x} = \boldsymbol{L}(G)^\dagger \boldsymbol{b}$ by satisfying the following condition

$$\|\boldsymbol{x} - \widetilde{\boldsymbol{x}}\|_{\boldsymbol{L}(G)} \leq \varepsilon \|\boldsymbol{x}\|_{\boldsymbol{L}(G)}, \tag{2.16}$$

where $\|\boldsymbol{x}\|_{\boldsymbol{L}(G)} := \sqrt{\boldsymbol{x}^T \boldsymbol{L}(G)\boldsymbol{x}}$.

Each iteration of the Conjugate Gradient method boils down to performing one multiplication of a vector by the matrix $\boldsymbol{L}(G)$, thus, as $\boldsymbol{L}(G)$ has $O(m)$ non-zero entries, the resulting running time is $O(m\sqrt{\kappa_f(\boldsymbol{L}(G))}\log 1/\varepsilon)$.[5]

As $\kappa_f(\boldsymbol{L}(G))$ can still be fairly large, an idea that allows reducing the running time of Conjugate Gradient method is preconditioning. Roughly speaking, a *precon-*

[5]One can also show that, as long as one is using exact arithmetic, Conjugate Gradient method will always converge to an *exact* solution after at most $O(n)$ iterations (independently of the value of $\kappa_f(\boldsymbol{L}(G))$). The resulting running time of $O(mn)$ is, however, still prohibitive from our point of view.

ditioner of a matrix A is a matrix B that provides good spectral approximation of A and enables us to solve a linear system with respect to B fast. Having such a preconditioner, we can employ it to guide the iterative steps of the Conjugate Gradient method to obtain faster convergence.

More precisely, it is known that if one is solving a Laplacian system in matrix $\boldsymbol{L}(G)$ and take some positive semi-definite matrix $\boldsymbol{A}$ as a preconditioner, the Preconditioned Conjugate Gradient method allows one to obtain an ε-approximate solution to this Laplacian system after at most $O(\sqrt{\kappa_f(\boldsymbol{A}, \boldsymbol{L}(G))} \log 1/\varepsilon)$ iterations, where $\kappa_f(\boldsymbol{A}, \boldsymbol{L}(G))$ denotes the *relative finite condition number* equal to the finite condition number $\kappa_f(\boldsymbol{A}^\dagger \boldsymbol{L}(G))$ of the matrix $\boldsymbol{A}^\dagger \boldsymbol{L}(G)$, and each of these iterations requires solving a linear system in $\boldsymbol{A}$.

Note that one can view the original Conjugate Gradient method as a special case of the preconditioned version. If A is an identity matrix $\boldsymbol{I}$ then $\kappa_f(\boldsymbol{I}, \boldsymbol{L}(G)) = \kappa_f(\boldsymbol{L}(G))$ and solving linear system in $\boldsymbol{I}$ is trivial. It turns out, however, that using non-trivial preconditioners allows for a dramatically faster (approximate) solving of Laplacian systems.

In particular, inspired by the pioneering work of Vaidya [135], Spielman and Teng [128, 129] (see also [130]) designed a Laplacian system solver that employs Preconditioned Conjugate Gradient method (with appropriate choice of preconditioners) and works in *nearly-linear time*, i.e., it provides an ε-approximate solution in time $\widetilde{O}(m \log 1/\varepsilon)$. This solver was, in turn, simplified and sped-up recently by Koutis, Miller and Peng [96] to provide the following result.

Theorem 2.7.1 ([96]). *For any $\varepsilon > 0$, and Laplacian system $\boldsymbol{L}(G)\boldsymbol{x} = \boldsymbol{b}$ with $\boldsymbol{b}$ being in range of $\boldsymbol{L}(G)$, one can obtain an ε-approximate solution $\widetilde{\boldsymbol{x}}$ to it in time $O(m \log^2 n (\log \log n)^2 \log 1/\varepsilon) = \widetilde{O}(m \log 1/\varepsilon)$.*

Note that the running time of the above solver is within a small poly-logarithmic factor of the obvious lower bound of $\Omega(m)$ and its running time is comparable to the running time of such elementary algorithms as Dijkstra's algorithm. As we will see later, such an extremely fast solver constitutes a powerful primitive that will enable us to make progress on a several basic graph problems.

The key idea behind the construction of the fast Laplacian system solver from Theorem 2.7.1, is building preconditioners for a given graph Laplacian $\boldsymbol{L}(G)$ from Laplacians of appropriately chosen (and reweighted) subgraphs of the graph G. This idea was first proposed by Vaidya [135] and then was fully developed by Spielman and Teng in [129]. This graph-based preconditioners that were introduced by Spielman and Teng are called *ultrasparsifiers*.

Ultrasparsifiers: Formally, an *k-ultrasparsifier* $\widetilde{G}$ of G, for any parameter $k \geq 1$, is a (reweighted) subgraph of G that is extremely sparse – i.e., it consists of a spanning tree of G and a small, $\widetilde{O}(n/k)$, number of additional edges – and preserves the spectral picture of G pretty well, that is, the corresponding relative finite condition number $\kappa_f(\boldsymbol{L}(\widetilde{G}), \boldsymbol{L}(G))$ is at most k.

The construction of the k-ultrasparsifier due to Spielman and Teng [129] is based on finding first a low-stretch spanning tree (as in Theorem 2.6.3) of G for an appropriate choice of weights. Next, this tree is augmented with a sufficiently large number of additional edges to ensure that the relative condition number of the Laplacian of the resulting graph (with respect to $\boldsymbol{L}(G)$) is at most $k/2$. Finally, by applying spectral sparsification (as in Theorem 2.6.1) to each dense cluster formed by these additional edges, one can ensure that the total number of them becomes $\widetilde{O}(n/k)$ and the relative condition number is at most k, as desired.

Now, roughly speaking, the way the Laplacian systems solver of Spielman and Teng works is as follows. By taking k to be large enough (but still poly-logarithmic) and applying the Preconditioned Conjugate Gradient method with the corresponding k-ultrasparsifier $\widetilde{G}$ of G, one essentially reduces the task of (ε-approximately) solving the Laplacian system corresponding to $\boldsymbol{L}(G)$, to solving a small number of Laplacian systems in the matrix $\boldsymbol{L}(\widetilde{G})$. If the choice of k was large enough, $\widetilde{G}$ will be a graph with only $n - 1 + o(n)$ edges, so a significant fraction of its vertices has to have a degree of 1 or 2. These vertices can be eliminated (via Cholesky factorization) to make the size of the Laplacian system one needs to solve significantly smaller. By solving this resulting system recursively, one obtains the desired nearly-linear time Laplacian systems solver (see [129] for details).

2.8 Packing Flows via Multiplicative-Weights-Update Method

A task that will arise in many of the results presented in this thesis is packing into a given capacitated graph G a convex combination of flows coming from some set $\mathcal{F}$.

Formally, given a capacitated graph $G = (V, E, \boldsymbol{u})$, and a set of flows $\mathcal{F}$, we are looking for coefficients $\{\lambda_f\}_{f\in\mathcal{F}}$, where each λ_f corresponds to a $f \in \mathcal{F}$, that satisfy the following set of constraints that we will call $(\mathcal{F}, G)$-*system*:

$$\begin{aligned} \forall_{e\in E} \sum_{f\in\mathcal{F}} \lambda_f |f(e)| &\leq u_e \\ \sum_{f\in\mathcal{F}} \lambda_f &= 1 \\ \forall_{f\in\mathcal{F}} \quad \lambda_f &\geq 0. \end{aligned} \tag{2.17}$$

(Note that, a priori, we do not bound the number of λ_fs – in our applications, however, only a small number of them will be non-zero.)

To make the edge-constraints of the above system more homogeneous, let us define for a given flow f and an edge e, the *congestion* $\mathsf{cong}_f(e)$ *of* e *(with respect to* f*)* to be $\frac{|f(e)|}{u_e}$. Then, we can rewrite (2.17) in the following form

$$\begin{aligned} \forall_{e\in E} \sum_{f\in\mathcal{F}} \lambda_f \mathsf{cong}_f(e) &\leq 1 \\ \sum_{f\in\mathcal{F}} \lambda_f &= 1 \\ \forall_{f\in\mathcal{F}} \quad \lambda_f &\geq 0. \end{aligned} \tag{2.18}$$

To illustrate why having a solution to the above system might be useful, consider $\mathcal{F}$ to be a set of all s-t flows in G of some prescribed value F. Now, if $\{\lambda_f^*\}_f$ would be a solution to this system then we could associate with it a convex combination of flows $\sum_f \lambda_f^* f$. Note that this convex combination corresponds naturally to a valid s-t flow $\overline{f}$ with $\overline{f}(e) = \sum_f \lambda_f^* f(e)$ for any $e \in E$. This flow would have a value of F and, by definition, would need to be feasible, i.e., respect all the capacities of the graph G. So, it would be a witness that shows that the maximum value F^* attainable by a feasible s-t flow in G has to be at least F. On the other hand, if the system of constraints would have no solution, this would mean that $F > F^*$ (otherwise the solution corresponding to just taking the maximum s-t flow f^* would satisfy the constraints). We can therefore see that the ability to solve such $(\mathcal{F}, G)$-systems (or deciding that they are infeasible), for various values of F, together with employment of binary search, would allow us to solve the maximum s-t flow problem.

Unfortunately, in practice, it is hard to obtain fast methods of solving such systems exactly. Therefore, in this thesis, we will be settling for approximate solutions. Namely, for a given $\gamma \geq 1$, let us define a γ-*relaxed* version of a $(\mathcal{F}, G)$-system to be

$$\begin{aligned} \forall_{e\in E} \sum_{f\in\mathcal{F}} \lambda_f \mathsf{cong}_f(e) &\leq \gamma \\ \sum_{f\in\mathcal{F}} \lambda_f &= 1 \\ \forall_{f\in\mathcal{F}} \quad \lambda_f &\geq 0. \end{aligned} \tag{2.19}$$

Now, our goal will be to solve such $(\mathcal{F}, G)$-systems approximately. Formally, we would like to obtain algorithms of the following type.

Definition 2.8.1. *For a given set of flows $\mathcal{F}$, graph G, and $\gamma \geq 1$, we define a* γ-approximate solver for $(\mathcal{F}, G)$-system *to be a procedure that:*

- *if the system is feasible (i.e., all the constraints of* (2.18) *can be satisfied), it finds a feasible solution to its γ-relaxed version (cf.* (2.19)*);*
- *on the other hand, if the system is infeasible, it either finds a feasible solution to the γ-relaxed system* (2.19) *as above, or* ***"fail"*** *is returned to indicate the infeasibility of this system.*

To put this definition in context, note that having such a γ-approximate solver for the above-mentioned case of the s-t flows would enable us to solve the maximum s-t problem γ-approximately.

2.8.1 Designing a γ-approximate Solver

Let us fix some capacitated graph $G = (V, E, \boldsymbol{u})$, a set of flows $\mathcal{F}$ and the corresponding $(\mathcal{F}, G)$-system. Our γ-approximate solver for this system will be based on, so called, multiplicative-weights-update method. The multiplicative-weights-update method is a framework established by Arora, Hazan and Kale [14] to encompass proof techniques employed in [113, 139, 65, 70]. In our setting, one can understand the approach of this framework as a way of taking an algorithm that solves given $(\mathcal{F}, G)$-system very crudely and, by calling it repeatedly, converting it into an algorithm that provides a pretty good solution to this system. The crude algorithm is treated as a black-box, so it can be thought of as an oracle that answers a certain type of queries.

More precisely, instead of solving the $(\mathcal{F}, G)$-system directly, this oracle is solving the following, much simpler, task. It is given a set of weights $\{w_e\}_e$ – one weight for each edge-constraint – and its task is to output a flow $f \in \mathcal{F}$ that respects the edge-constraints only *on weighted average*, where weighting in this average is given by the weights $\{w_e\}_e$.

To formalize this notion, we introduce the following definition.

Definition 2.8.2. *Given an $\alpha \geq 1$ and some $(\mathcal{F}, G)$-system, by an α-oracle for this system we mean an algorithm that given weights vector $\boldsymbol{w}$ returns:*

- *A flow $f \in \mathcal{F}$ such that*

$$\sum_e w_e \mathsf{cong}_f(e) \leq \alpha \sum_e w_e = \alpha |\boldsymbol{w}|_1,$$

  if the $(\mathcal{F}, G)$-system is feasible;

- ***"fail"** or a flow $f \in \mathcal{F}$ as above, otherwise*

Remark: An important detail of the above definition is that sometime the oracle might return the desired flow f even though the $(\mathcal{F}, G)$-system is infeasible. To understand why this might be the case, one should observe that our requirements on f are weaker than the requirements on a feasible solution to the $(\mathcal{F}, G)$-system.

Note that if $f^* \in \mathcal{F}$ is a feasible solution to the $(\mathcal{F}, G)$-system then we must have that $\sum_e w_e \mathsf{cong}_{f^*}(e) \leq |\boldsymbol{w}|_1$. So, whenever the system is feasible there exists at least one flow that satisfies the condition required of the oracle.

Now, the way we will use an α-oracle to (approximately) solve the corresponding $(\mathcal{F}, G)$-system is by employing the routine presented in Figure 2-1. This routine takes an α-oracle O and an accuracy parameter $\delta > 0$ as an input, then it presents this oracle with a sequence of queries corresponding to different weight vectors. Finally, it either combines the obtained answers to output a γ-approximate solution to the $(\mathcal{F}, G)$-system, where $\gamma = \alpha(1 + 3\delta)$, or it returns **"fail"** to indicate that the system is not feasible.

In a bit greater detail, the way such a γ-approximate solution is build is by maintaining weights for each edge-constraint – initially, all the weights are set to 1. In

Input : An accuracy parameter $1/2 > \delta > 0$, capacitated graph $G = (V, E, \boldsymbol{u})$, corresponding $(\mathcal{F}, G)$-system, and an α-oracle O for it;
Output: Either a convex combination $\{\lambda_f\}_{f \in \mathcal{F}}$ of flows from $\mathcal{F}$, or **"fail"** indicating that the $(\mathcal{F}, G)$-system is infeasible;

Initialize $w_e^0 \leftarrow 1$ for all edges e, $i = 1$, and $N \leftarrow \frac{2 \ln m}{\delta^2}$
while $\sum_{i'=1}^{i-1} \overline{\lambda}^{i'} < 1$ **do**
  Query the oracle O with edge weights given by $\boldsymbol{w}^{i-1}$
  if O returns **"fail" then return "fail"**
  else
    Let f^i be the returned flow and let $\rho^i \leftarrow \max_{e \in E} \mathsf{cong}_{f^i}(e)$ be the maximum congestion f^i exerts in G
    Set $\overline{\lambda}^i \leftarrow \min\{\frac{1}{N\rho^i}, 1 - \sum_{i'=1}^{i-1} \overline{\lambda}^i\}$
    Set $w_e^i \leftarrow w_e^{i-1}(1 + \delta N \overline{\lambda}^i \mathsf{cong}_{f^i}(e))$ for each $e \in E$
    Increment i
  end
end
Let $\lambda_f \leftarrow \sum_{i: f^i = f} \overline{\lambda}^i$, for each $f \in \mathcal{F}$
return $\{\lambda_f\}_f$

Figure 2-1: Multiplicative-weights-update routine

each iteration i, the oracle O is called with these weights. If this call returns a flow $f^i \in \mathcal{F}$, the weight of each edge is multiplied by a factor of $(1+\delta N \overline{\lambda}^i \mathsf{cong}_{f^i}(e))$, where $N\overline{\lambda}^i$ is roughly proportional to the inverse of the largest congestion ρ^i exerted by f^i in G. This iterative process finishes once the sum of all the $\overline{\lambda}^i$ coefficients becomes one.

The key fact underlying the above process is that if in some iteration i the congestion of some edge is poor, say close to ρ^i, then the weight of the corresponding edge-constraint will increase by a factor of roughly $(1 + \delta)$. On the other hand, if congestion of an edge is good, e.g., the flow on that edge is no more than its capacity, then the weight of the corresponding edge-constraint is essentially unchanged. In this way, we make sure that a larger fraction of the total weight is being put on the violated constraints, so the oracle is being forced to return solutions that come closer to satisfying these constraints (possibly at the expense of some other, previously only slightly violated, constraints).

Now, the dynamics that emerges from this process is that, on one hand, the total weight $|\boldsymbol{w}|_1$ does not grow too quickly, due to the average (weighted) congestion constraint on the flows returned by the oracle O (cf. Definition 2.8.2). On the other hand, if a particular edge e consistently suffers large congestion in a sequence of flows returned by O then the weight of corresponding edge-constraint increases rapidly relative to the total weight, which significantly penalizes any further congestion of this edge in the subsequent flows. So, as in the end we are returning an appropriately weighted convex combination of all the obtained flows, this ensures that no edge-

constraint is violated by more than γ in it.

We proceed to making the above intuition precise by proving that our routine will indeed return the desired γ-approximate solution to the $(\mathcal{F}, G)$-system. After that, in section 2.8.3, we will analyze how by looking at some additional properties of the oracle one can bound the number of oracle calls this routine makes before termination – this will be critical to bounding the total running time of the resulting algorithms.

2.8.2 Correctness of the Multiplicative-Weights-Update Routine

In this section, we prove the correctness of the routine from Figure 2-1 by establishing the following theorem.

Theorem 2.8.3. *For any $1/2 > \delta > 0$ and $\alpha \geq 1$, if all the oracle calls in the routine from Figure 2-1 terminate without returning **"fail"**, the convex combination $\{\lambda_f\}_f$ returned is an γ-approximate solution to the corresponding $(\mathcal{F}, G)$-system, with $\gamma = \alpha(1 + 3\delta)$.*

Our proof of the above theorem will be based on the potential function $\mu_i := |\boldsymbol{w}^i|_1$. Clearly, $\mu_0 = m$ and this potential only increases during the course of the algorithm. Now, the crucial fact we will be using is that from Definition 2.8.2 it follows that if the oracle O did not return **"fail"** then

$$\sum_e w_e^{i-1} \mathsf{cong}_{f^i}(e) \leq \alpha |\boldsymbol{w}^{i-1}|_1, \tag{2.20}$$

for all $i \geq 1$.

We start the proof by upper bounding the total growth of μ_i throughout the algorithm.

Lemma 2.8.4. *For any $i \geq 0$,*

$$\mu_{i+1} \leq \mu_i \exp\left(\alpha\delta N \overline{\lambda}^{i+1}\right).$$

In particular, after the final, T-th iteration, the total weight $|\boldsymbol{w}^T|_1 = \mu_T$ is at most $m \exp\left(\alpha\delta N \sum_{i=1}^{T} \overline{\lambda}^i\right) = n^{O(\alpha/\delta)}$.

Proof. For any $i \geq 0$, we have

$$\begin{aligned}
\mu_{i+1} = \sum_e w_e^{i+1} &= \sum_e w_e^i \left(1 + \delta N \overline{\lambda}^{i+1} \mathsf{cong}_{f^{i+1}}(e)\right) \\
&= \sum_e w_e^i + \delta N \overline{\lambda}^{i+1} \sum_e w_e^i \mathsf{cong}_{f^{i+1}}(e) \\
&\leq \mu_i + \alpha\delta N \overline{\lambda}^{i+1} |\boldsymbol{w}^i|_1,
\end{aligned}$$

where the last inequality follows from (2.20). Thus, we can conclude that

$$\mu_{i+1} \leq \mu_i + \alpha\delta N\overline{\lambda}^{i+1}|\boldsymbol{w}^i|_1 = \mu_i(1+\alpha\delta N\overline{\lambda}^{i+1}) \leq \mu_i \exp\left(\alpha\delta N\overline{\lambda}^{i+1}\right),$$

as desired. The lemma follows. □

One of the consequences of the above lemma is that whenever we make a call to the oracle, the total weight $|\boldsymbol{w}^i|_1$ is at most $n^{O(\alpha/\delta)}$. Thus, we have an absolute bound on the value of the weights the oracle is dealing with.

Next, we tie the final weight w_e^T of a particular edge e to the congestion

$$\sum_{f\in\mathcal{F}} \lambda_f \mathsf{cong}_f(e)$$

that this edge suffers with respect to the convex combination output by the algorithm.

Lemma 2.8.5. *For any edge e and $i \geq 0$,*

$$w_e^i \geq \exp\left((1-\delta)\delta N \sum_{i'=1}^{i} \overline{\lambda}^{i'} \mathsf{cong}_{f^{i'}}(e)\right).$$

In particular, for the last, T-th iteration, we have

$$w_e^T \geq \exp\left((1-\delta)\delta N \sum_{i'=1}^{T} \overline{\lambda}^{i'} \mathsf{cong}_{f^{i'}}(e)\right) = \exp\left((1-\delta)\delta N \sum_{f\in\mathcal{F}} \lambda_f \mathsf{cong}_f(e)\right).$$

Proof. For any $i \geq 0$, we have

$$w_e^i = \prod_{i'=1}^{i}\left(1+\delta N\overline{\lambda}^{i'}\mathsf{cong}_{f^{i'}}(e)\right) \geq \prod_{i'=1}^{i}\exp\left((1-\delta)\delta N\overline{\lambda}^{i'}\mathsf{cong}_{f^{i'}}(e)\right),$$

where we used the fact that, by definition, $N\overline{\lambda}^{i}\mathsf{cong}_{f^i}(e) \leq 1$ and that for any $1/2 > \delta > 0$ and $x \in [0,1]$:

$$\exp((1-\delta)\delta x) \leq (1-\delta x).$$

Now, the lemma follows since

$$w_e^i \geq \prod_{i'=1}^{i}\exp\left((1-\delta)\delta N\overline{\lambda}^{i'}\mathsf{cong}_{f^{i'}}(e)\right) = \exp\left((1-\delta)\delta N\sum_{i'=1}^{i}\overline{\lambda}^{i'}\mathsf{cong}_{f^{i'}}(e)\right),$$

as desired. □

Finally, by Lemmas 2.8.4 and 2.8.5, we conclude that for any edge e,

$$m\exp(\alpha\delta N) \geq \mu_T = |\boldsymbol{w}^T|_1 \geq w_e^T \geq \exp\left((1-\delta)\delta N\sum_{f\in\mathcal{F}}\lambda_f\mathsf{cong}_f(e)\right).$$

This implies that

$$\sum_{f\in\mathcal{F}} \lambda_f \mathsf{cong}_f(e) \leq \frac{\alpha}{(1-\delta)} + \frac{\ln m}{(1-\delta)\delta N} = \frac{\alpha}{(1-\delta)} + \frac{\delta}{2(1-\delta)} \leq \alpha(1+3\delta)$$

for every edge e, as $1/2 > \delta > 0$. The Theorem 2.8.3 follows.

2.8.3 Convergence of the Multiplicative-Weights-Update Routine

Note that even though we proved above that our multiplicative-weights-update routine from Figure 2-1 will always returned the desired solution, it is unclear how many iterations will this routine make before termination. We proceed now to relating this number of iterations to some properties of the oracle that is employed.

Width-based bound: Probably the simplest and most broadly applied way of bounding the number of iterations is based on the concept of width of an oracle.

Definition 2.8.6. *For a given oracle O for some $(\mathcal{F}, G)$-system, we say that O has* width ρ *if for any possible weights $\boldsymbol{w}$, the flow returned by O in response to them exerts a congestion of at most ρ in G.*

Clearly, width of an oracle provides us with a simple way of bounding the number of queries to it, as explain in the following lemma.

Lemma 2.8.7. *If the oracle O used by the routine from Figure 2-1 has width ρ then at most $O(\rho\delta^{-2}\log m)$ iterations is made.*

Proof. By Definition 2.8.6, we must have that all $\rho^i \leq \rho$ and thus each $\overline{\lambda}^i$ (possibly except $\overline{\lambda}^T$) has to be at least $\frac{1}{N\rho^i} \geq \frac{1}{N\rho}$. This means that $\sum_{i'=1}^{i} \overline{\lambda}^i$ reaches 1 after at most $N\rho + 1$ iterations. The lemma follows. □

Tightness-based bound: Even though the use of width is a very simple way of bounding the running time of the multiplicative-weights-update routine, sometime one can obtain a better bound. This is possible as the width-based bound is, in some sense, pessimistic, i.e., it does not give a good handle on situations in which only a few of the calls made in the multiplicative-weights-update routine results in flows with ρ^i being close to the width ρ.

To get a different type of bound that avoids this problem, let us define $\rho_G(f)$ to be the maximum congestion $\max_{e\in E} \mathsf{cong}_f(e)$ that flow f exerts in G. (Note that if f is returned by an oracle with width ρ then $\rho_G(f)$ has to be at most ρ.) Also, let $\kappa_G(f) := \{e \in E \mid \mathsf{cong}_f(e) \geq \frac{\rho_G(f)}{2}\}$ be the set of edges of G whose congestion is within a factor of two of the maximum congestion exerted by f in G.

Definition 2.8.8. *For a given oracle O for some $(\mathcal{F}, G)$-system, we say that O has* tightness k *if for any possible weights $\boldsymbol{w}$, the flow f returned by O in response to it is such that $|\kappa_G(f)| \geq k$.*

Observe that, trivially, every oracle has a tightness of at least one.

Now, we prove the following lemma that tells us that the number of oracle calls in our multiplicative-weights-update routine is inversely proportional to the tightness of the oracle.

Lemma 2.8.9. *If the α-oracle O used by the routine from Figure 2-1 has tightness k then at most $\frac{2\alpha Nm}{k} = \frac{4\alpha m \log m}{k\delta^2}$ iterations is made.*

Proof. Consider the following potential function:

$$\phi^i = \sum_{e \in E} \sum_{i'=1} i\overline{\lambda}^{i'} \mathsf{cong}_{f^{i'}}(e).$$

Note that $\phi^0 = 0$, $\phi^{i+1} \geq \phi^i$ for each i, and, by Theorem 2.8.3, we know that

$$\sum_{i'=1}^{T} \overline{\lambda}^{i'} \mathsf{cong}_{f^{i'}}(e) = \sum_{f \in \mathcal{F}} \lambda_f \mathsf{cong}_f(e) \leq \gamma = \alpha(1+3\delta),$$

for each edge e. Therefore, the final value ϕ^T of this potential is at most $\alpha(1+3\delta)m$.

Now, observe that in each iteration i of the multiplicative-weights-update routing from Figure 2-1, we have that for every $e \in \kappa_G(f^i)$,

$$\overline{\lambda}^{i} \mathsf{cong}_{f^{i}}(e) \geq \frac{1}{2N}.$$

As a result, we see that in each iteration the potential ϕ increases by at least $\frac{k}{2N}$. This implies that the total number of iterations is at most $\frac{2\alpha(1+3\delta)mN}{k}$ and the lemma follows. □

Note that the bounds offered by Lemma 2.8.7 and Lemma 2.8.9 are in general incomparable. Therefore, in this thesis, we will be using one or the other depending on the characteristic of the flow packing problem we will be dealing with.

Part I

Cuts and Flows

Chapter 3

Approximation of Maximum Flow in Undirected Graphs

In this chapter, we consider the maximum s-t flow problem and the dual problem of finding a minimum s-t cut.[1] We propose a new approach to solving these problems approximately in a capacitated, undirected graph. Our approach is based on solving a sequence of electrical flow problems. Each electrical flow is given by a solution to a Laplacian system (cf. Section 2.4) and thus may be approximately computed in nearly-linear time (cf. Section 2.7).

Using this approach, we develop the fastest known algorithm for computing approximately maximum s-t flow. For a graph having n vertices and m edges, our algorithm computes a $(1-\varepsilon)$-approximately maximum s-t flow in time[2] $\widetilde{O}(mn^{1/3}\varepsilon^{-11/3})$. A dual version of our approach gives the fastest known algorithm for computing a $(1+\varepsilon)$-approximately minimum s-t cut. Its running time is $\widetilde{O}(m+n^{4/3}\varepsilon^{-16/3})$.

3.1 Introduction

The maximum s-t flow problem and its dual, the minimum s-t cut problem, are two of the most fundamental and extensively studied problems in Operations Research and Optimization [121, 2]. They have many applications (see [3]) and are often used as subroutines in other algorithms (see e.g. [14, 123]). Also, the celebrated Max-Flow Min-Cut theorem [67, 57] that establishes equality of the value of the maximum s-t flow and capacity of the minimum s-t cut, is a prototypical example of duality relation in optimization. Many advances have been made in the development of algorithms for this problem (see Goldberg and Rao [73] for an overview). However, for the basic problem of computing or $(1-\varepsilon)$-approximating the maximum flow in undirected, unit-capacity graphs with $m = O(n)$ edges, the asymptotically fastest known algorithm is the one developed in 1975 by Even and Tarjan [61], which takes time $O(n^{3/2})$. Despite 35 years of extensive work on the problem, this bound has not been improved.

[1] This chapter is based on joint work with Paul Christiano, Jonathan Kelner, Daniel Spielman, and Shang-Hua Teng and contains material from [40].

[2] Recall that $\widetilde{O}(g(m))$ denotes $O(g(m)\log^c g(m))$ for some constant c.

In this chapter, we introduce a new approach to computing approximately maximum s-t flows and minimum s-t cuts in undirected, capacitated graphs. Using it, we present the first algorithms that break the $O(n^{3/2})$ complexity barrier described above. In addition to being the fastest known algorithms for these problems, they are simple to describe and introduce techniques that may be applicable to other tasks. One of the key steps of these algorithms is reducing the problem of computing maximum flows subject to capacity constraints to the problem of computing electrical flows in resistor networks. An approximate solution to each electrical flow problem can be found in time $\widetilde{O}(m)$ using recently developed algorithms for solving systems of linear equations in Laplacian matrices [96, 130] that are discussed in Section 2.7.

There is a simple physical intuition that underlies our approach, which we describe here in the case of a graph with unit edge capacities. We begin by thinking of each edge of the input graph as a resistor with resistance one, and we compute the electrical flow that results when we send current from the source s to the sink t. These currents obey the flow conservation constraints, but they are ignorant of capacities of the edges. To remedy this, we increase the resistance of each edge in proportion to the amount of current flowing through it—thereby penalizing edges that violate their capacities—and compute the electrical flow with these new resistances.

After repeating this operation $\widetilde{O}(m^{1/3} \cdot \mathrm{poly}(1/\varepsilon))$ times, we will be able to obtain a $(1-\varepsilon)$-approximately maximum s-t flow by taking a certain average of the electrical flows that we have computed, and we will be able to extract a $(1+\varepsilon)$-approximately minimum s-t cut from the vertex potentials[3]. This will give us algorithms for both problems that run in time $\widetilde{O}(m^{4/3} \cdot \mathrm{poly}(1/\varepsilon))$. By combining this with the graph smoothing and sampling techniques of Karger [87], we can get a $(1-\varepsilon)$-approximately maximum s-t flow in time $\widetilde{O}(mn^{1/3}\varepsilon^{-11/3})$. Furthermore, by applying the cut algorithm to a sparsifier [26] of the input graph, we can compute a $(1+\varepsilon)$-approximately minimum s-t cut in time $\widetilde{O}(m + n^{4/3}\varepsilon^{-16/3})$.

We remark that the results presented in this chapter immediately improve the running time of algorithms that use the computation of an approximately maximum s-t flow on an undirected, capacitated graph as a subroutine. For example, combining our work with that of Sherman [123] allows us to achieve the best currently known asymptotic approximation ratio of $O(\sqrt{\log n})$ for the sparsest cut problem in time $O(m + n^{4/3+\delta})$ for any constant $\delta > 0$.

3.1.1 Previous Work on Maximum Flows and Minimum Cuts

The best previously known algorithms for the problems studied here are obtained by combining techniques of Goldberg and Rao [73] and Benczúr and Karger [27]. In a breakthrough paper, Goldberg and Rao [73] developed an algorithm for computing exact maximum s-t flows in directed or undirected capacitated graphs in time

$$O(m \min(n^{2/3}, m^{1/2}) \log(n^2/m) \log U),$$

[3]For clarity, we will analyze these two cases separately, and they will use slightly different rules for updating the resistances.

assuming that the edge capacities are integers between 1 and U. In undirected graphs, one can find faster approximation algortithms for these problems by using the graph sparsification techniques of Benczúr and Karger [26, 27]. Goldberg and Rao [73] observe that by running their exact maximum flow algorithm on the sparsifiers of Benczúr and Karger [26], one can obtain a $(1+\varepsilon)$-approximately minimum cut in time $\widetilde{O}(m + n^{3/2}\varepsilon^{-3})$. This provides an approximation of the value of the maximum flow. To actually obtain a feasible flow whose value approximates the maximum one can apply the algorithm of Goldberg and Rao in the divide-and-conquer framework of Benczúr and Karger [27]. This provides a $(1-\varepsilon)$-approximately maximum flow in time $\widetilde{O}(m\sqrt{n}\varepsilon^{-1})$. We refer the reader to the the paper of Goldberg and Rao for a survey of the previous work on algorithms for computing maximum s-t flows.

In more recent work, Daitch and Spielman [51] showed that fast solvers for Laplacian linear systems [130, 96] could be used to make interior-point algorithms for the maximum flow and minimum-cost flow problems run in time $\widetilde{O}(m^{3/2}\log U)$, and, as we will see in Chapter 5, one can approximate a wide range of cut problems, including the minimum s-t cut problem, within a polylogarithmic factor in almost linear time.

3.1.2 Outline of This Chapter

We begin in Section 3.2 by introducing some notations, as well as, stating some simple facts that we will need in the sequel. In Section 3.3 we give a simplified version of our approximate maximum-flow algorithm that has running time of $\widetilde{O}(m^{3/2}\varepsilon^{-5/2})$.

In Section 3.4, we will show how to improve the running time of our algorithm to $\widetilde{O}(m^{4/3}\varepsilon^{-3})$; we will then describe how to combine this with existing graph smoothing and sparsification techniques to compute approximately maximum s-t flows in time $\widetilde{O}(mn^{1/3}\varepsilon^{-11/3})$ and to approximate the value of such flows in time $\widetilde{O}(m+n^{4/3}\varepsilon^{-16/3})$. In Section 3.5, we present a variant of our algorithm that computes approximately minimum s-t cuts in time $\widetilde{O}(m + n^{4/3}\varepsilon^{-16/3})$.

3.2 Preliminaries and Notations

Following the convention introduced in Section 2.1, let us denote by $G = (V, E, \boldsymbol{u})$ an undirected capacitated graph with n vertices and m edges. We distinguish two vertices, a *source* vertex s and a *sink* vertex t. Each edge e has a nonzero integral capacity u_e, and we let $U := \max_e u_e / \min_e u_e$ be the ratio of the largest to the smallest capacities.

Now, the *minimum s-t cut problem* is that of finding an s-t cut in G of minimum capacity.[4] On the other hand, the *maximum s-t flow problem* is that of finding a feasible s-t flow in G of maximum value. We denote a maximum flow in G by f^*, and we denote its value by $F^* := |f^*|$. We say that f is a $(1-\varepsilon)$-approximately maximum flow if it is a feasible s-t flow of value at least $(1-\varepsilon)F^*$. Similarly, we say that C is

[4]The reader can consult Section 2.1 for some of the definitions that are not stated here.

an $(1+\varepsilon)$-approximately minimum s-t cut if its capacity is within $(1+\varepsilon)$ factor of the capacity of the minimum cut C^*.

To simplify the exposition, we will take ε to be a constant independent of m throughout this chapter, and m will be assumed to be larger than some fixed constant. However, our analysis will go through unchanged as long as $\varepsilon > \widetilde{\Omega}(m^{-1/3})$. In particular, our analysis will apply to all ε for which our given bounds are faster than the $O(m^{3/2})$ time required by existing exact algorithms.

One can easily reduce the problem of finding a $(1-\varepsilon)$-approximation to the maximum flow in an arbitrary undirected graph to that of finding a $(1-\varepsilon/2)$-approximation in a graph in which the ratio of the largest to smallest capacities is polynomially bounded. To do this, one should first compute a crude approximation of the maximum flow in the original graph. For example, one can compute the s-t path of maximum bottleneck in time $O(m + n \log n)$ [121, Section 8.6e], where we recall that the bottleneck of a path is the minimum capacity of an edge on that path. If this maximum bottleneck of an s-t path is B, then the maximum flow lies between B and mB. This means that there is a maximum flow in which each edge flows at most mB, so all capacities can be decreased to be at most mB. On the other hand, if one removes all edges with capacities less than $\varepsilon B/2m$, the maximum flow can decrease by at most $\varepsilon B/2$. So, we can assume that the minimum capacity is at least $\varepsilon B/2m$ and the maximum is at most Bm, for a ratio of $2m^2/\varepsilon$. Thus, by a simple scaling, we can assume that all edge capacities are integers between 1 and $2m^2/\varepsilon$.

3.3 A Simple $\tilde{O}(m^{3/2}\varepsilon^{-5/2})$-Time Flow Algorithm

Before describing our $\widetilde{O}(m^{4/3}\varepsilon^{-3})$ algorithm, we will describe a simpler algorithm that finds a $(1-O(\varepsilon))$-approximately maximum flow in time $\widetilde{O}(m^{3/2}\varepsilon^{-5/2})$. Our final algorithm will be obtained by carefully modifying the one described here.

The algorithm will employ the flow packing framework we described in Section 2.8. To see how this framework can be applied, consider $\mathcal{F}$ being the set of all the s-t flows in G of value F, for some value F. Note that if one obtains an $(1+\varepsilon)$-approximate solution $\{\lambda_f\}_f$ to the corresponding $(\mathcal{F}, G)$-system then taking a flow $\overline{f}$ being just a convex combination of the flows indicated by these λ_fs, provides us with an s-t flow of value F that exerts a congestion of at most $(1+\varepsilon)$ in G. Such a flow $\overline{f}$ thus certifies that the value F^* of the maximum s-t flow in G is at least $F/(1+\varepsilon)$. On the other hand, if the $(\mathcal{F}, G)$-system is infeasible, then it means that the value of F^* has to be smaller than F.

Now, by our discussion in Section 3.2, we know that the value F^* of the maximum s-t flow in G lies between B and mB, where B is the maximum bottleneck of an s-t path that we can compute fast. Therefore, we can use binary search to turn an $(1+\varepsilon)$-approximate solver for a $(\mathcal{F}, G)$-system mentioned above into a $(1-O(\varepsilon))$-approximation algorithm for the maximum s-t flow problem that we are seeking. Note that as our lower and upper bound on F^* is within a factor m of each other, the running time overhead introduced by doing this is $O(\log m/\varepsilon)$ and thus negligible.

In the light of the above, we can focus from now on on a particular value of F and

design a $(1+\varepsilon)$-approximate solver for the corresponding $(\mathcal{F}, G)$-system. This will be done by employing the multiplicative-weights-update routine from Figure 2-1 with accuracy parameter δ equal to ε. To this end, we will design a simple $(1+\varepsilon)$-oracle for the $(\mathcal{F}, G)$-system that has a width of $3\sqrt{m/\varepsilon}$ and runs in $\widetilde{O}(m \log \varepsilon^{-1})$ time. By our discussion above, Theorem 2.8.3 and Lemma 2.8.7, providing such an oracle will give the desired $(1 - O(\varepsilon))$-approximation algorithm for the maximum s-t flow problem running in time $\widetilde{O}(m^{3/2}\varepsilon^{-5/2})$.

3.3.1 From Electrical Flows to Maximum Flows: Oracle Construction

The key component of our oracle will be an approximate electrical flow computations using the fast Laplacian solvers discussed in Section 2.7. More precisely, we will use the following theorem, whose proof is deferred to Section 3.3.2.

Theorem 3.3.1 (Fast Approximation of Electrical Flows)**.** *For any $\delta > 0$, any $F > 0$, and any vector $\boldsymbol{r}$ of resistances in which the ratio of the largest to the smallest resistance is at most R, we can compute, in time $\widetilde{O}(m \log R/\delta)$, a vector of vertex potentials $\tilde{\boldsymbol{\phi}}$ and an s-t flow $\tilde{f}$ of value F such that*

a. $\mathcal{E}_{\boldsymbol{r}}(\tilde{f}) \leq (1+\delta)\mathcal{E}_{\boldsymbol{r}}(f)$, and $\mathcal{E}_{\boldsymbol{r}}(\tilde{f}) \leq (1+\delta)\mathcal{E}_{\boldsymbol{r}}(f)$ where f is the electrical s-t flow of value F,

b. for every edge e,

$$\left| r_e f(e)^2 - r_e \tilde{f}(e)^2 \right| \leq \frac{\delta}{2mR} \mathcal{E}_{\boldsymbol{r}}(f),$$

where f is the true electrical flow.

c.

$$\widetilde{\phi}_s - \widetilde{\phi}_t \geq \left(1 - \frac{\delta}{12nmR}\right) F R_{\text{eff}}^{s,t}(\boldsymbol{r}).$$

We will refer to a flow meeting the above conditions as a δ-approximate electrical flow*.*

Now, to build our oracle, we set

$$r_e := \frac{1}{u_e^2}\left(w_e + \frac{\varepsilon |\boldsymbol{w}|_1}{3m}\right) \tag{3.1}$$

for each edge e, and we use the procedure from Theorem 3.3.1 to approximate the electrical flow that sends F units of flow from s to t in a network whose resistances are given by the r_e. The pseudocode for this oracle is shown in Figure 3-1.

It is easy to see that $\tilde{f}$ is an s-t flow of value F and thus $\tilde{f} \in \mathcal{F}$. So, to established the desired properties of our oracle we need to show that the resulting flow $\tilde{f}$ meets also the conditions required by Definition 2.8.2 (with $\alpha = (1+\varepsilon)$) and Definition 2.8.6. We will do it by using the basic fact that, by its definition, the electrical flows

Input : A graph $G = (V, E)$ with capacities $\{u_e\}_e$, a target flow value F, and edge weights $\{w_e\}_e$
Output: Either a flow $\tilde{f}$, or **"fail"** indicating that $F > F^*$

$r_e \leftarrow \frac{1}{u_e^2}\left(w_e + \frac{\varepsilon|\boldsymbol{w}|_1}{3m}\right)$ for each $e \in E$
Find an $(\varepsilon/3)$-approximate electrical flow $\tilde{f}$ using Theorem 3.3.1 on G with resistances $\boldsymbol{r}$ and target flow value F
if $\mathcal{E}_{\boldsymbol{r}}(\tilde{f}) > (1+\varepsilon)|\boldsymbol{w}|_1$ **then return** ***"fail"***
else return $\tilde{f}$

Figure 3-1: A simple implementation of an $(1+\varepsilon)$-oracle with width $3\sqrt{m/\varepsilon}$

minimize the energy of the flow. Our analysis will then be based on comparing the energy of $\tilde{f}$ with that of an optimal max flow f^*. Intuitively, what will turn out is that the w_e term in equation (3.1) guarantees the bound on the average congestion from Definition 2.8.2, while the $\varepsilon|\boldsymbol{w}|_1/(3m)$ term guarantees the bound on the width, i.e., on the maximum congestion $\rho_G(\tilde{f})$ exerted by $\tilde{f}$.

To prove it formally, suppose f^* is a maximum flow. By its feasibility, we have $\mathsf{cong}_{f^*}(e) \leq 1$ for all e, so

$$\begin{aligned}
\mathcal{E}_{\boldsymbol{r}}(f^*) &= \sum_e \left(w_e + \frac{\varepsilon|\boldsymbol{w}|_1}{3m}\right)\left(\frac{f^*(e)}{u_e}\right)^2 \\
&= \sum_e \left(w_e + \frac{\varepsilon|\boldsymbol{w}|_1}{3m}\right)\left(\mathsf{cong}_{f^*}(e)\right)^2 \\
&\leq \sum_e \left(w_e + \frac{\varepsilon|\boldsymbol{w}|_1}{3m}\right) \\
&= \left(1 + \frac{\varepsilon}{3}\right)|\boldsymbol{w}|_1.
\end{aligned}$$

Since the electrical flow minimizes the energy, $\mathcal{E}_{\boldsymbol{r}}(f^*)$ is an upper bound on the energy of the electrical flow of value F whenever $F \leq F^*$. In this case, Theorem 3.3.1 implies that our $(\varepsilon/3)$-approximate electrical flow satisfies

$$\mathcal{E}_{\boldsymbol{r}}(\tilde{f}) \leq \left(1 + \frac{\varepsilon}{3}\right)\mathcal{E}_{\boldsymbol{r}}(f^*) \leq \left(1 + \frac{\varepsilon}{3}\right)^2 |\boldsymbol{w}|_1 \leq (1+\varepsilon)\,|\boldsymbol{w}|_1. \tag{3.2}$$

This shows that our oracle will never output **"fail"** when $F \leq F^*$. It thus suffices to show that the energy bound $\mathcal{E}_{\boldsymbol{r}}(\tilde{f}) > (1+\varepsilon)|\boldsymbol{w}|_1$, which holds whenever the algorithm does not return **"fail"**, implies the required bounds on the average and worst-case congestion. To see this, we note that the energy bound implies that

$$\sum_e w_e \left(\mathsf{cong}_{\tilde{f}}(e)\right)^2 \leq (1+\varepsilon)\,|\boldsymbol{w}|_1, \tag{3.3}$$

and, for all $e \in E$,

$$\frac{\varepsilon |\boldsymbol{w}|_1}{3m} \left(\mathsf{cong}_{\tilde{f}}(e)\right)^2 \leq (1+\varepsilon) |\boldsymbol{w}|_1. \tag{3.4}$$

By the Cauchy-Schwarz inequality,

$$\left(\sum_e w_e \mathsf{cong}_{\tilde{f}}(e)\right)^2 \leq |\boldsymbol{w}|_1 \left(\sum_e w_e \left(\mathsf{cong}_{\tilde{f}}(e)\right)^2\right), \tag{3.5}$$

so equation (3.3) gives us that

$$\sum_e w_e \mathsf{cong}_{\tilde{f}}(e) \leq \sqrt{1+\varepsilon}\, |\boldsymbol{w}|_1 < (1+\varepsilon) |\boldsymbol{w}|_1, \tag{3.6}$$

which is the required bound on the average congestion. Furthermore, equation (3.4) and the fact that $\varepsilon < 1/2$ implies that

$$\mathsf{cong}_{\hat{f}}(e) \leq \sqrt{\frac{3m(1+\varepsilon)}{\varepsilon}} \leq 3\sqrt{m/\varepsilon}$$

for all e, which establishes the required bound on the maximum congestion. So our algorithm indeed implements an $(1+\varepsilon)$-oracle with width $3\sqrt{m/\varepsilon}$), as desired.

To bound the running time of this oracle, recall that we can assume that all edge capacities lie between 1 and $U = m^2/\varepsilon$ and obtain

$$R = \max_{e,e'} \frac{r_e}{r_{e'}} \leq U^2 \frac{\max_e r_e + \varepsilon |\boldsymbol{w}|_1}{\varepsilon |\boldsymbol{w}|_1} \leq U^2 \frac{m+\varepsilon}{\varepsilon} = O\left((m/\varepsilon)^{O(1)}\right). \tag{3.7}$$

This establishes an upper bound on the ratio of the largest resistance to the smallest resistance. Thus, by Theorem 3.3.1, the running time of this implementation is $\widetilde{O}(m \log R/\varepsilon) = \widetilde{O}(m \log 1/\varepsilon)$. Combining this with Theorem 2.8.3 and Lemma 2.8.7, we get

Theorem 3.3.2. *For any $0 < \varepsilon < 1/2$, the maximum flow problem can be $(1-\varepsilon)$-approximated in $\widetilde{O}(m^{3/2}\varepsilon^{-5/2})$ time.*

3.3.2 Computing Electrical Flows

In this section, we prove Theorem 3.3.1.

Proof of Theorem 3.3.1. Without loss of generality, we may scale the resistances so that they lie between 1 and R. This gives the following relation between F and the energy of the electrical s-t flow of value F:

$$\frac{F^2}{m} \leq \mathcal{E}_{\boldsymbol{r}}(f) \leq F^2 R m. \tag{3.8}$$

Let $\boldsymbol{L}$ be the Laplacian matrix for this electrical flow problem, i.e., one in which weights of edges are equal to the inverses of their resistances. Note that all off-

diagonal entries of $\boldsymbol{L}$ lie between -1 and $-1/R$. Recall that the vector of optimal vertex potentials $\boldsymbol{\phi}$ of the electrical flow is the solution to the following linear system:

$$\boldsymbol{L}\boldsymbol{\phi} = F\boldsymbol{\chi}_{s,t}$$

Using Theorem 2.7.1, for any $\varepsilon > 0$,we can compute in time

$$O(m \log^2 n (\log\log n)^2 \log 1/\varepsilon) = \widetilde{O}(m \log 1/\varepsilon)$$

a set of potentials $\widehat{\boldsymbol{\phi}}$ (cf. (2.16)) such that

$$\left\| \widehat{\boldsymbol{\phi}} - \boldsymbol{\phi} \right\|_{\boldsymbol{L}} \leq \varepsilon \left\| \boldsymbol{\phi} \right\|_{\boldsymbol{L}},$$

where

$$\left\| \boldsymbol{\phi} \right\|_{\boldsymbol{L}} \stackrel{\text{def}}{=} \sqrt{\boldsymbol{\phi}^T \boldsymbol{L} \boldsymbol{\phi}}.$$

Note also that

$$\left\| \boldsymbol{\phi} \right\|_{\boldsymbol{L}}^2 = \mathcal{E}_{\boldsymbol{r}}(f),$$

where f is the potential flow obtain from $\boldsymbol{\phi}$ via Ohm's law, i.e., via equation 2.7 (and it corresponds exactly to the electrical s-t flow of value F). Now, let $\hat{f}$ be the potential flow obtained from $\widehat{\boldsymbol{\phi}}$, its vector representation is given by

$$\hat{\boldsymbol{f}} = \boldsymbol{C}\boldsymbol{B}^T \widehat{\boldsymbol{\phi}}.$$

The flow $\hat{f}$ is not necessarily an s-t flow because $\widehat{\boldsymbol{\phi}}$ only approximately satisfies the linear system. Therefore, there could be some small surpluses or deficits of the flow at vertices other than s and t. In other words, the vector representation $\hat{\boldsymbol{f}}$ of $\hat{f}$ may not satisfy the equation $\boldsymbol{B}\hat{\boldsymbol{f}} = F\boldsymbol{\chi}_{s,t}$ (cf. equation (2.8)). The way we address this problem will be computing an approximation $\tilde{f}$ of $\hat{f}$ that such that its vector representation $\tilde{\boldsymbol{f}}$ satisfies $\boldsymbol{B}\tilde{\boldsymbol{f}} = F\boldsymbol{\chi}_{s,t}$.

Before fixing this problem, we first observe that the energy of $\hat{f}$ is not much more than the energy of f. We have

$$\left\| \widehat{\boldsymbol{\phi}} \right\|_{\boldsymbol{L}}^2 = \mathcal{E}_{\boldsymbol{r}}(\hat{f}),$$

and by the triangle inequality

$$\left\| \widehat{\boldsymbol{\phi}} \right\|_{\boldsymbol{L}} \leq \left\| \boldsymbol{\phi} \right\|_{\boldsymbol{L}} + \left\| \widehat{\boldsymbol{\phi}} - \boldsymbol{\phi} \right\|_{\boldsymbol{L}} \leq (1+\varepsilon) \left\| \boldsymbol{\phi} \right\|_{\boldsymbol{L}}.$$

So

$$\mathcal{E}_{\boldsymbol{r}}(\hat{f}) \leq (1+\varepsilon)^2 \mathcal{E}_{\boldsymbol{r}}(f).$$

To see that the flow $\hat{f}$ does not flow too much into or out of any vertex other than

s and t, note that the flows into and out of vertices are given by the vector

$$\boldsymbol{i}_{ext} \stackrel{\text{def}}{=} \boldsymbol{B}\hat{\boldsymbol{f}}.$$

Observe that the sum of the entries in $\boldsymbol{i}_{ext}$ is zero. Furthermore, we will produce an s-t flow of value F by adding a flow to $\hat{f}$ to obtain a flow $\tilde{f}$ for which

$$\boldsymbol{B}\tilde{\boldsymbol{f}} = F\boldsymbol{\chi}_{s,t}.$$

So, if η is the maximum discrepancy between $\boldsymbol{i}_{ext}$ and $F\boldsymbol{\chi}_{s,t}$:

$$\eta \stackrel{\text{def}}{=} \|\boldsymbol{i}_{ext} - F\boldsymbol{\chi}_{s,t}\|_\infty .$$

then we have

$$\eta \le \|\boldsymbol{i}_{ext} - F\boldsymbol{\chi}_{s,t}\|_2 = \left\|\boldsymbol{L}\widehat{\boldsymbol{\phi}} - \boldsymbol{L}\boldsymbol{\phi}\right\|_2 \le \|\boldsymbol{L}\|_2 \left\|\widehat{\boldsymbol{\phi}} - \boldsymbol{\phi}\right\|_{\boldsymbol{L}} \le 2n \left\|\widehat{\boldsymbol{\phi}} - \boldsymbol{\phi}\right\|_{\boldsymbol{L}} \le 2n\varepsilon\sqrt{\mathcal{E}_{\boldsymbol{r}}(f)}.$$

It is an easy matter to produce an s-t flow $\tilde{f}$ that differs from the flow represented by $\hat{f}$ by at most $n\eta$ on each edge. For example, one can do this by solving a flow problem in a spanning tree of the graph. Let T be any spanning tree of the graph G. We now construct a flow in T that with the demands $F\boldsymbol{\chi}_{s,t}(u) - \boldsymbol{i}_{ext}(u)$ at each vertex u. As the sum of the positive demands in this flow is at most $n\eta$, one can find such a flow in which every edge flows at most $n\eta$. Moreover, since the edges of T are a tree, one can find the flow in linear time. We obtain the flow $\tilde{f}$ by adding this flow to the flow $\hat{f}$. The resulting flow is an s-t flow of value F. Moreover,

$$\left\|\hat{\boldsymbol{f}} - \tilde{\boldsymbol{f}}\right\|_\infty \le n\eta.$$

To ensure that we get a good approximation of the electrical flow, we now impose the requirement that

$$\varepsilon \le \frac{\delta}{2n^2 m^{1/2} R^{3/2}}.$$

To check that requirement a is satisfied, observe that

$$\begin{aligned}
\mathcal{E}_{\boldsymbol{r}}(\tilde{f}) &= \sum_e r_e \tilde{f}_e^2 \\
&\le \sum_e r_e (\hat{f}_e + n\eta)^2 \\
&\le \mathcal{E}_{\boldsymbol{r}}(\hat{f}) + 2n\eta \sum_e r_e \hat{f}_e + n^2\eta^2 \sum_e r_e \\
&\le \mathcal{E}_{\boldsymbol{r}}(\hat{f}) + (2n\eta F + n^2\eta^2) \sum_e r_e \\
&\le \mathcal{E}_{\boldsymbol{r}}(\hat{f}) + \varepsilon(2n^2(mR)^{1/2} + 4n^4)\mathcal{E}_{\boldsymbol{r}}(f)mR \qquad \text{(by (3.8))} \\
&\le \mathcal{E}_{\boldsymbol{r}}(f)\left((1+\varepsilon)^2 + \varepsilon 6n^4 mR^{3/2}\right).
\end{aligned}$$

We may ensure that this is at most $(1+\delta)\mathcal{E}_{\boldsymbol{r}}(f)$ by setting

$$\varepsilon = \frac{\delta}{12n^4 m R^{3/2}}.$$

Now, to establish part b, note that

$$\|\boldsymbol{\phi}\|_{\boldsymbol{L}} = \sum_e r_e f_e^2$$

and so

$$\sum_e r_e (f_e - \hat{f}_e)^2 = \left\|\boldsymbol{\phi} - \widehat{\boldsymbol{\phi}}\right\|_{\boldsymbol{L}}^2 \leq \varepsilon^2 \mathcal{E}_{\boldsymbol{r}}(f).$$

We now bound $\left|r_e f_e^2 - r_e \hat{f}_e^2\right|$ by

$$\begin{aligned}\left|r_e f_e^2 - r_e \hat{f}_e^2\right|^2 &= r_e \left(f_e - \hat{f}_e\right)^2 r_e \left(f_e + \hat{f}_e\right)^2 \\ &\leq \varepsilon^2 \mathcal{E}_{\boldsymbol{r}}(f)(2+\varepsilon)^2 \mathcal{E}_{\boldsymbol{r}}(f),\end{aligned}$$

which implies

$$\left|r_e f_e^2 - r_e \hat{f}_e^2\right| \leq \varepsilon(2+\varepsilon)\mathcal{E}_{\boldsymbol{r}}(f).$$

It remains to bound $\left|r_e \hat{f}_e^2 - r_e \tilde{f}_e^2\right|$, which we do by the calculation

$$\begin{aligned}\left|r_e \hat{f}_e^2 - r_e \tilde{f}_e^2\right| &\leq \sqrt{R}\left|\hat{f}_e - \tilde{f}_e\right| \sqrt{r_e}\left|\hat{f}_e + \tilde{f}_e\right| \\ &\leq \sqrt{R} 2n^2 \varepsilon \sqrt{\mathcal{E}_{\boldsymbol{r}}(f)}(2+\varepsilon)\sqrt{\mathcal{E}_{\boldsymbol{r}}(f)} \\ &= \varepsilon\sqrt{R} 2(2+\varepsilon) n^2 \mathcal{E}_{\boldsymbol{r}}(f).\end{aligned}$$

Putting these inequalities together we find

$$\left|r_e f_e^2 - r_e \tilde{f}_e^2\right| \leq \varepsilon\sqrt{R}(2n^2+1)(2+\varepsilon)\mathcal{E}_{\boldsymbol{r}}(f) \leq \mathcal{E}_{\boldsymbol{r}}(f)\frac{\delta}{2mR}.$$

To prove part c, first recall that

$$\boldsymbol{L}\boldsymbol{\phi} = F\boldsymbol{\chi}_{\boldsymbol{s},\boldsymbol{t}}$$

and that

$$\mathcal{E}_{\boldsymbol{r}}(f) = F^2 R_{\text{eff}}(\boldsymbol{r})$$

So,

$$\begin{aligned}
\varepsilon^2 \|\boldsymbol{\phi}\|_{\boldsymbol{L}}^2 &\geq \left\|\boldsymbol{\phi} - \tilde{\boldsymbol{\phi}}\right\|_{\boldsymbol{L}}^2 \\
&= \left(\boldsymbol{\phi} - \tilde{\boldsymbol{\phi}}\right)^T \boldsymbol{L}\left(\boldsymbol{\phi} - \tilde{\boldsymbol{\phi}}\right) \\
&= \left\|\tilde{\boldsymbol{\phi}}\right\|_{\boldsymbol{L}}^2 + \|\boldsymbol{\phi}\|_{\boldsymbol{L}}^2 - 2\tilde{\boldsymbol{\phi}}^T \boldsymbol{L}\boldsymbol{\phi} \\
&= \|\boldsymbol{\phi}\|_{\boldsymbol{L}}^2 + \left\|\tilde{\boldsymbol{\phi}}\right\|_{\boldsymbol{L}}^2 - 2F\boldsymbol{\chi}_{s,t}{}^T\tilde{\boldsymbol{\phi}}.
\end{aligned}$$

Thus,

$$\begin{aligned}
2F\boldsymbol{\chi}_{s,t}{}^T\tilde{\boldsymbol{\phi}} &\geq \|\boldsymbol{\phi}\|_{\boldsymbol{L}}^2 + \left\|\tilde{\boldsymbol{\phi}}\right\|_{\boldsymbol{L}}^2 - \varepsilon^2 \|\boldsymbol{\phi}\|_{\boldsymbol{L}}^2 \\
&\geq \left(1 + (1-\varepsilon)^2 - \varepsilon^2\right) \|\boldsymbol{\phi}\|_{\boldsymbol{L}}^2 \\
&= (2 - 2\varepsilon) \|\boldsymbol{\phi}\|_{\boldsymbol{L}}^2 \\
&= (2 - 2\varepsilon) F^2 R_{\text{eff}}(\boldsymbol{r})
\end{aligned}$$

By dividing both side of the obtained inequality by $2F$ and noting that $\boldsymbol{\chi}_{s,t}{}^T\tilde{\boldsymbol{\phi}} = \widetilde{\phi}_s - \widetilde{\phi}_t$

□

3.4 An $\widetilde{O}(mn^{1/3}\varepsilon^{-11/3})$ Algorithm for Approximate Maximum Flow

In this section, we modify our algorithm to run in time $\widetilde{O}(m^{4/3}\varepsilon^{-3})$. We then combine this with the smoothing and sampling techniques of Karger [87] to obtain an $\widetilde{O}(mn^{1/3}\varepsilon^{-11/3})$-time algorithm.

For fixed ε, the algorithm in the previous section required us to compute $\widetilde{O}(m^{1/2})$ electrical flows, each of which took time $\widetilde{O}(m)$, which,in turn, led to a running time of $\widetilde{O}(m^{3/2})$. To reduce this running time to $\widetilde{O}(m^{4/3})$, we'll show how to find an approximately maximum s-t flow while computing only $\widetilde{O}(m^{1/3})$ electrical flows.

Before we proceed to doing this, let us recall that our analysis of the oracle from Section 3.3.1 was fairly simplistic, and therefore one might hope to improve the running time of the algorithm by proving a tighter bound on the width. Unfortunately, the graph in Figure 3-2 shows that our analysis was essentially tight. The graph consists of k parallel paths of length k connecting s to t, along with a single edge e that directly connects s to t. The max flow in this graph is $k+1$. In the first call made to the oracle by the multiplicative weights routine, all of the edges will have the same resistance. In this case, the electrical flow of value $k+1$ will send $(k+1)/2k$ units of flow along each of the k paths and $(k+1)/2$ units of flow across e. Since the graph has $m = \Theta(k^2)$, the width of the oracle in this case is $\Theta(m^{1/2})$.

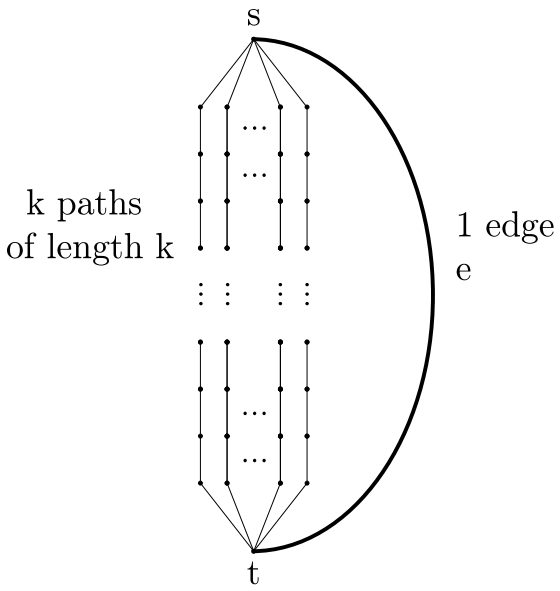


Figure 3-2: A graph on which the electrical flow sends approximately $\sqrt{m}$ units of flow across an edge when sending the maximum flow F^* from s to t.

3.4.1 The Improved Algorithm

The example from Figure 3-2 that we discussed above, shows that it is possible for the electrical flow returned by the oracle to exceed the edge capacities by $\Theta(m^{1/2})$. However, we note that if one removes the edge e from the graph in Figure 3-2, the electrical flow on the resulting graph is much better behaved, but the value of the maximum flow is only very slightly reduced. This demonstrates a phenomenon that will be central to our improved algorithm: while instances in which the electrical flow sends a huge amount of flow over some edges exist, they are somewhat fragile, and they are often drastically improved by removing the bad edges.

This motivates us to modify our algorithm as follows. We'll set ρ to be some value smaller than the actual worst-case bound of $\widetilde{O}(m^{1/2})$. (It will end up being $\widetilde{O}(m^{1/3})$.) The oracle will begin by computing an electrical flow as before. However, when this electrical flow exceeds the capacity of some edge e by a factor greater than ρ, we'll remove e from the graph and try again, keeping all of the other weights the same. We'll repeat this process until we obtain a flow in which all edges flow at most a factor of ρ times their capacity (or some failure condition is reached), and we'll use this flow (as an oracle answer) in our multiplicative-weights-update routine. All the edges removed by the oracle during this process, are added to a set H of *forbidden edges*. We will keep these edges permanently removed from the graph, i.e., they will not be included in the graphs supplied to future invocations of the oracle.

In Figures 3-3, we present the modified versions of the oracle implementation, where we have highlighted the parts that have changed from the simpler version shown in Figures 3-1. The set H is initialized to be empty and then is passed between consecutive oracle calls.

Input : A graph $G = (V, E)$ with capacities $\{u_e\}_e$, a target flow value F, edge weights $\{w_e\}_e$, and a set H of forbidden edges

Output: Either a flow $\tilde{f}$ and a set H of forbidden edges, or **"fail"** indicating that $F > F^*$

$\rho \leftarrow \frac{8m^{1/3}\ln^{1/3} m}{\varepsilon}$

$r_e \leftarrow \frac{1}{u_e^2}\left(w_e + \frac{\varepsilon|\boldsymbol{w}|_1}{3m}\right)$ for each $e \in E \setminus H$

Find an approximate electrical flow $\tilde{f}$ using Theorem 3.3.1 on $G_H := (V, E \setminus H)$ with resistances $\boldsymbol{r}$, target flow value F, and parameter $\delta = \varepsilon/3$.

if $\mathcal{E}_{\boldsymbol{r}}(\tilde{f}) > (1+\varepsilon)|\boldsymbol{w}|_1$ or s and t are disconnected in G_H **then return** ***"fail"***

if there exists e with $\text{cong}_{\tilde{f}}(e) > \rho$ **then** add e to H and start over

return $\tilde{f}$

Figure 3-3: The modified oracle O' used by our improved algorithm

3.4.2 Analysis of the New Algorithm

Before proceeding to a formal analysis of the new algorithm, it will be helpful to examine what is already guaranteed by the analysis from Section 3.3, and what we'll need to show to demonstrate the algorithm's correctness and bound its running time.

We first note that, by construction, the congestion of any edge in the flow $\tilde{f}$ returned by the modified oracle from Figure 3-3 will be bounded by ρ. Furthermore, as this oracle enforces the bound $\mathcal{E}_{\boldsymbol{r}}(\tilde{f}) \leq (1+\varepsilon)|\boldsymbol{w}|_1$; by equations (3.3), (3.5), and (3.6) in Section 3.3.1, this guarantees that $\tilde{f}$ will meet the weighted average congestion bound required for a $(1+\varepsilon)$-oracle for the $(\mathcal{F}, G)$-system of our interest. So, as long as the modified oracle always successfully returns a flow, it will function as an $(1+\varepsilon)$-oracle with width ρ, and our analysis from Section 3.3 will show that the multiplicative-weights-update routine employed by our algorithm (with $\delta = \varepsilon$) will yield a $(1+O(\varepsilon))$-approximately maximum s-t flow after $\widetilde{O}(\rho\varepsilon^{-2})$ iterations.

Our problem is thus reduced to understanding the behavior of the modified oracle. To prove correctness, we will need to show that whenever the modified oracle is called and the value F is at most F^*, it will return some flow $\tilde{f}$ (as opposed to returning **"fail"**). To bound the running time, we will need to provide an upper bound on the total number of electrical flows computed by the modified oracle throughout the execution of the algorithm.

To this end, we will show the following upper bound on the cardinality $|H|$ and the capacity $u(H)$ of the set of forbidden edges. The proof of these bounds is postponed until the next section.

Lemma 3.4.1. *Throughout the execution of the algorithm,*

$$|H| \leq \frac{30m\ln m}{\varepsilon^2\rho^2}$$

and

$$u(H) \leq \frac{30mF \ln m}{\varepsilon^2 \rho^3}.$$

Now, if we plug in the value $\rho = (8m^{1/3} \ln^{1/3} m)/\varepsilon$ used by the algorithm, the above lemma gives us that $|H| \leq \frac{15}{32}(m \ln m)^{1/3}$ and $u(H) \leq \frac{15}{256}\varepsilon F < \varepsilon F/12$.

Given these bounds, it is now straightforward to show the following theorem, which establishes the correctness and bounds the running time of our algorithm.

Theorem 3.4.2. *For any $0 < \varepsilon < 1/2$, if $F \leq F^*$ our improved algorithm will return a feasible s-t flow $\overline{f}$ of value $|\overline{f}| = (1 - O(\varepsilon))F$ in time $\widetilde{O}(m^{4/3}\varepsilon^{-3})$.*

Proof. To bound the running time, we note that whenever we invoke the algorithm from Theorem 3.3.1, we either advance the number of iterations of the multiplicative-weights-update routine or increase the cardinality of the set H. As a result, the number of linear systems we solve is at most $\widetilde{O}(\rho\varepsilon^{-2}) + |H| \leq \widetilde{O}(\rho\varepsilon^{-2}) + \frac{15}{32}(m \ln m)^{1/3}$.

equation (3.7) implies that the value of R from Theorem 3.3.1 is $O((m/\varepsilon)^{O(1)})$, so solving each linear system takes time at most $\widetilde{O}(m \log 1/\varepsilon)$. This gives an overall running time of

$$\widetilde{O}((\rho\varepsilon^{-2} + \tfrac{15}{32}(m \ln m)^{1/3})m) = \widetilde{O}(m^{4/3}\varepsilon^{-3}),$$

as claimed.

It thus remains to prove correctness. For this, we need to show that if $F \leq F^*$, then the oracle does not return **"fail"**. By definition of the oracle, this can occur only if we disconnect s from t or if $\mathcal{E}_{\boldsymbol{r}}(\tilde{f}) > (1+\varepsilon)|\boldsymbol{w}|_1$. By Lemma 3.4.1 and the comment following it, we know that throughout the whole algorithm G_H has maximum flow value of at least $F^* - \varepsilon F/12 \geq (1 - \varepsilon/12)\, F$ and thus, in particular, we will never disconnect s from t.

Furthermore, this implies that there exists a feasible flow in our graph of value $(1 - \varepsilon/12)\, F$, even after we have removed the edges in H. There is thus a flow of value F in which every edge has congestion at most $1/(1 - \varepsilon/12)$. We can therefore use the argument from Section 3.3.1 (equation (3.2) and the lines directly preceding it) to show that we always have

$$\mathcal{E}_{\boldsymbol{r}}(\tilde{f}) \leq (1 + \varepsilon/12)^2(1 + \varepsilon/3)^2|\boldsymbol{w}|_1 \leq (1 + \varepsilon)|\boldsymbol{w}|_1,$$

as required. □

The above theorem allows us to apply the binary search strategy that we used in Section 3.3. This yields the following theorem:

Theorem 3.4.3. *For any $0 < \varepsilon < 1/2$, the maximum flow problem can be $(1 - \varepsilon)$-approximated in $\widetilde{O}(m^{4/3}\varepsilon^{-3})$ time.*

3.4.3 The Proof of Lemma 3.4.1

All that remains is to prove the bounds given by Lemma 3.4.1 on the cardinality and capacity of H. To do so, we will use the effective resistance of the circuits on

which we compute electrical flows. The key insight is that we only cut an edge when its flow accounts for a nontrivial fraction of the energy of the electrical flow, and that cutting such an edge will cause a substantial change in the effective resistance. Combining this with a bound on how much the effective resistance can change during the execution of the algorithm will guarantee that we won't cut too many edges.

Before we do all of that, let us first prove the following lemma which gives a lower bound on the effect that increasing the resistance of an edge can have on the effective resistance.

Lemma 3.4.4. *Let f be an electrical s-t flow on a graph G with resistances $\boldsymbol{r}$. Suppose that some edge $h = (i, j)$ accounts for a β fraction of the total energy of f, i.e.,*

$$f(h)^2 r_h = \beta \mathcal{E}_{\boldsymbol{r}}(f).$$

For some $\gamma > 0$, define new resistances $\boldsymbol{r}'$ such that $r'_h = \gamma r_h$, and $r'_e = r_e$ for all $e \neq h$. Then

$$R_{\text{eff}}(\boldsymbol{r}') \geq \frac{\gamma}{\beta + \gamma(1-\beta)} R_{\text{eff}}(\boldsymbol{r}).$$

In particular:

- *If we "cut" the edge h by setting $\gamma = \infty$, then*

$$R_{\text{eff}}(\boldsymbol{r}') \geq \frac{R_{\text{eff}}(\boldsymbol{r})}{1-\beta}.$$

- *If we slightly increase the effective resistance of the edge h by setting $\gamma = (1+\varepsilon)$ with $\varepsilon \leq 1$, then*

$$R_{\text{eff}}(\boldsymbol{r}') \geq \frac{1+\varepsilon}{\beta + (1+\varepsilon)(1-\beta)} R_{\text{eff}}(\boldsymbol{r}) \geq \left(1 + \frac{\varepsilon\beta}{2}\right) R_{\text{eff}}(\boldsymbol{r}).$$

Proof. The assumptions of the theorem are unchanged if we multiply f by a constant, so we may assume without loss of generality that f is the electrical s-t flow of value $1/R_{\text{eff}}(\boldsymbol{r})$. If $\boldsymbol{\phi}^f$ is the vector of vertex potentials corresponding to f, this gives $\phi^f_s - \phi^f_t = 1$. Since adding a constant to the potentials doesn't change the flow, we may assume that $\phi^f_s = 1$ and $\phi^f_t = 0$. By Fact 2.4.1,

$$C_{\text{eff}}(\boldsymbol{r}) = \sum_{(u,v)\in E} \frac{\left(\phi^f_u - \phi^f_v\right)^2}{r_{(u,v)}} \quad = \quad \frac{\left(\phi^f_i - \phi^f_j\right)^2}{r_h} + \sum_{(u,v)\in E\setminus\{h\}} \frac{\left(\phi^f_u - \phi^f_v\right)^2}{r_{(u,v)}}.$$

The assumption that h contributes a β fraction of the total energy implies that, in the above expression,

$$\frac{\left(\phi^f_i - \phi^f_j\right)^2}{r_h} = \beta C_{\text{eff}}(\boldsymbol{r})$$

and thus

$$\sum_{(u,v)\in E\setminus\{h\}} \frac{\left(\phi_u^f - \phi_v^f\right)^2}{r_{(u,v)}} = (1-\beta)C_{\text{eff}}(\boldsymbol{r}).$$

We will obtain our bound on $C_{\text{eff}}(\boldsymbol{r}')$ by plugging the original vector of potentials ϕ^f into the expression in Fact 2.4.1:

$$\begin{aligned} C_{\text{eff}}(\boldsymbol{r}') &= \min_{\boldsymbol{\phi} \mid \substack{\phi_s=1,\\ \phi_t=0}} \sum_{(u,v)\in E} \frac{\left(\phi_u - \phi_v\right)^2}{r'_{(u,v)}} \quad \leq \quad \sum_{(u,v)\in E} \frac{\left(\phi_u^f - \phi_v^f\right)^2}{r'_{(u,v)}} \\ &= \frac{\left(\phi_i^f - \phi_j^f\right)^2}{r'_h} + \sum_{(u,v)\in E\setminus\{h\}} \frac{\left(\phi_u^f - \phi_v^f\right)^2}{r'_{(u,v)}} \quad = \quad \frac{\left(\phi_i^f - \phi_j^f\right)^2}{\gamma r_h} + \sum_{(u,v)\in E\setminus\{h\}} \frac{\left(\phi_u^f - \phi_v^f\right)^2}{r_{(u,v)}} \\ &= \frac{\beta}{\gamma} C_{\text{eff}}(\boldsymbol{r}) + (1-\beta)C_{\text{eff}}(\boldsymbol{r}) \quad = \quad C_{\text{eff}}(\boldsymbol{r}) \left(\frac{\beta + \gamma(1-\beta)}{\gamma}\right). \end{aligned}$$

Since $R_{\text{eff}}(\boldsymbol{r}) = 1/C_{\text{eff}}(\boldsymbol{r})$ and $R_{\text{eff}}(\boldsymbol{r}') = 1/C_{\text{eff}}(\boldsymbol{r}')$, the desired result follows. □

Now, to prove Lemma 3.4.1, let $\boldsymbol{r}^j$ be the resistances used in the j^{th} electrical flow computed during the execution of the algorithm. If an edge e is not in E or if e has been added to H by step j, set $r_e^j = \infty$. We define the potential function

$$\Phi(j) = R_{\text{eff}}(\boldsymbol{r}^j) = \mathcal{E}_{\boldsymbol{r}^j}(f^j),$$

where f^j is the (exact) electrical flow of value 1 arising from $\boldsymbol{r}^j$. Lemma 3.4.1 will follow easily from the following statement.

Lemma 3.4.5. *Suppose that $F \leq F^* \leq mF$. Then:*

(1) $\Phi(j)$ never decreases during the execution of the algorithm.

(2) $\Phi(1) \geq m^{-4}F^{-2}$.

(3) If we add an edge to H between steps $j-1$ and j, then $(1 - \frac{\varepsilon\rho^2}{5m})\Phi(j) > \Phi(j-1)$.

Proof. We prove each of the above properties one by one.

Proof of (1)

The only way that the resistance r_e^j of an edge e can change is if the weight w_e is increased by the multiplicative-weights-update routine, or if e is added to H so that r_e^j is set to ∞. As such, the resistances are nondecreasing during the execution of the algorithm. By Rayleigh Monotonicity (Corollary 2.4.2), this implies that the effective resistance is nondecreasing as well.

Proof of (2)

In the first linear system, $H = \emptyset$ and $r_e^1 = \frac{1+\varepsilon/3}{u_e^2}$ for all $e \in E$. Let C^* be the minimum s-t cut of G. By the Max Flow-Min Cut Theorem ([67, 57]), we know that the capacity $u(C^*)$ of this cut is equal to F^*. In particular,

$$r_e^1 = \frac{1+\varepsilon/3}{u_e^2} \geq \frac{1+\varepsilon/3}{F^{*2}} > \frac{1}{F^{*2}}$$

for all $e \in E(C^*)$. As f^1 is an electrical s-t flow of value 1, it sends 1 unit of net flow between C^* and $\overline{C}^*$; so, some edge $e' \in E(C^*)$ must have $f^1(e') \geq 1/m$. This gives

$$\Phi(1) = \mathcal{E}_{\boldsymbol{r}^1}(f^1) = \sum_{e \in E} f^1(e)^2 r_e^1 \geq f^1(e')^2 r_{e'}^1 > \frac{1}{m^2 F^{*2}}. \tag{3.9}$$

Since $F^* \leq mF$ by our assumption, the desired inequality follows.

Proof of (3)

Suppose we add the edge h to H between steps $j-1$ and j. We will show that h accounts for a substantial fraction of the total energy of the electrical flow with respect to the resistances $\boldsymbol{r}^{j-1}$, and use Lemma 3.4.4.

Now, let $\boldsymbol{w}$ be the weights used at step $j-1$, and let $\tilde{f}$ be the flow we computed in step $j-1$. Because we added h to H, we know that $\mathsf{cong}_{\tilde{f}}(h) > \rho$. Since our algorithm did not return **"fail"** after computing this $\tilde{f}$, we must have that

$$\mathcal{E}_{\boldsymbol{r}^{j-1}}(\tilde{f}) \leq (1+\varepsilon)|\boldsymbol{w}|_1. \tag{3.10}$$

Using the definition of r_h^{j-1}, the fact that $\mathsf{cong}_{\tilde{f}}(h) > \rho$, and equation (3.10), we obtain:

$$\begin{aligned}
\tilde{f}(h)^2 r_h^{j-1} &= \tilde{f}(h)^2 \frac{w_e + \varepsilon \frac{|\boldsymbol{w}|_1}{3m}}{u_e^2} \\
&\geq \tilde{f}(h)^2 \frac{\varepsilon |\boldsymbol{w}|_1}{3m u_e^2} \\
&= \frac{\varepsilon}{3m} \left(\frac{\tilde{f}(h)}{u_e} \right)^2 |\boldsymbol{w}|_1 \\
&= \frac{\varepsilon}{3(1+\varepsilon)m} \mathsf{cong}_{\tilde{f}}(h)^2 \left((1+\varepsilon)|\boldsymbol{w}|_1 \right) \\
&> \frac{\varepsilon \rho^2}{3(1+\varepsilon)m} \mathcal{E}_{\boldsymbol{r}^{j-1}}(\tilde{f}).
\end{aligned}$$

The above inequalities establish that edge h accounts for more than a $\frac{\varepsilon\rho^2}{3(1+\varepsilon)m}$ fraction of the total energy $\mathcal{E}_{\boldsymbol{r}^i}(\tilde{f})$ of the flow $\tilde{f}$.

The flow $\tilde{f}$ is the approximate electrical flow computed by our algorithm, but our argument will require that an inequality like this holds in the exact electrical flow f^{j-1}. This is guaranteed by part b of Theorem 3.3.1, which, along with the facts that $\mathcal{E}_{\boldsymbol{r}}(\tilde{f}) \geq \mathcal{E}_{\boldsymbol{r}}(f^{j-1})$, $\rho \leq 1$, and $\varepsilon < 1/2$, gives us that

$$\begin{aligned} f^{j-1}(h)^2 r_h^{j-1} > \tilde{f}(h)^2 r_h^{j-1} - \frac{\varepsilon/3}{2mR}\mathcal{E}_{\boldsymbol{r}}(f^{j-1}) &> \left(\frac{\varepsilon\rho^2}{3(1+\varepsilon)m} - \frac{\varepsilon/3}{2mR}\right)\mathcal{E}_{\boldsymbol{r}}(f^{j-1}) \\ &> \frac{\varepsilon\rho^2}{5m}\mathcal{E}_{\boldsymbol{r}}(f^{j-1}). \end{aligned}$$

The result now follows from Lemma 3.4.4. □

We are finally ready to prove Lemma 3.4.1.

Proof of Lemma 3.4.1. Let k be the cardinality of the set H at the end of the algorithm. Let $\tilde{f}$ be the flow that was produced by our algorithm just before k-th edge was added to H, let j be the time when this flow was output, and let $\boldsymbol{w}$ be the corresponding weights.

As the energy required by an s-t flow scales with the square of the value of the flow,

$$\Phi(j) \leq \frac{\mathcal{E}_{\boldsymbol{r}^j}(f)}{F^2} \leq \frac{\mathcal{E}_{\boldsymbol{r}^j}(\tilde{f})}{F^2}. \tag{3.11}$$

By the construction of our algorithm, it must have been the case that $\mathcal{E}_{\boldsymbol{r}^j}(\tilde{f}) \leq (1+\varepsilon)|\boldsymbol{w}|_1$. This inequality together with equation (3.11) and part 2 of Lemma 3.4.5 implies that

$$\Phi(j) = \mathcal{E}_{\boldsymbol{r}^j}(f^j) \leq \frac{\mathcal{E}_{\boldsymbol{r}^j}(\tilde{f})}{F^2} \leq (1+\varepsilon)|\boldsymbol{w}|_1 m^4 \Phi(1).$$

Now, since up to time j we had $k-1$ additions of edges to H, parts 1 and 3 of Lemma 3.4.5, and Lemma 2.8.4 imply that

$$\begin{aligned} \left(1 - \frac{\varepsilon\rho^2}{5m}\right)^{-(k-1)} \leq \frac{\Phi(j)}{\Phi(1)} \leq (1+\varepsilon)|\boldsymbol{w}|_1 m^4 &\leq (1+\varepsilon)m^4\left(m\exp\left(\frac{(1+\varepsilon)}{\varepsilon}\right)\right) \\ &\leq 2m^5\exp(3\varepsilon^{-1}\ln m), \end{aligned}$$

where the last inequality used the fact that $\varepsilon < 1/2$. Rearranging the terms in the above inequality gives us that

$$k \leq -\frac{\ln 2 + 5\ln m + 3\varepsilon^{-1}\ln m}{\ln\left(1 - \frac{\varepsilon\rho^2}{5m}\right)} + 1 < -\frac{6\varepsilon^{-1}\ln m}{\ln\left(1 - \frac{\varepsilon\rho^2}{5m}\right)} < \frac{30m\ln m}{\varepsilon^2\rho^2},$$

where we used the inequalities $\varepsilon < 1/2$ and $\log(1-c) < -c$ for all $c \in (0,1)$. This establishes our bound on cardinality of the set H.

To bound the value of $u(H)$, let us note that we add an edge e to H only when we send at least ρu_e units of flow across it. But since we never flow more than F units

across any single edge, we have that $u_e \leq F/\rho$. Therefore, we may conclude that

$$u(H) \leq |H|\frac{F}{\rho} \leq \frac{30mF \ln m}{\varepsilon^2 \rho^3},$$

as desired. □

3.4.4 Improving the Running Time to $\widetilde{O}(mn^{1/3}\varepsilon^{-11/3})$

We can now combine our algorithm with existing methods to further improve its running time. In [87] (see also [27]), Karger presented a technique, which he called "graph smoothing", that allows one to use random sampling to speed up an exact or $(1-\varepsilon)$-approximate flow algorithm. More precisely, his techniques yield the following theorem, which is implicit in [87] and stated in a more similar form in [27].

Theorem 3.4.6 ([87, 27])**.** *Let $T(m, n, \varepsilon)$ be the time needed to find a $(1-\varepsilon)$-approximately maximum flow in an undirected, capacitated graph with m edges and n vertices. Then one can obtain a $(1-\varepsilon)$-approximately maximal flow in such a graph in time $\widetilde{O}(\varepsilon^2 m/n \cdot T(\widetilde{O}(n\varepsilon^{-2}), n, \Omega(\varepsilon)))$.*

By applying the above theorem to our $\widetilde{O}(m^{4/3}\varepsilon^{-3})$ algorithm, we obtain our desired running time bound.

Theorem 3.4.7. *For any $0 < \varepsilon < 1/2$, the maximum flow problem can be $(1-\varepsilon)$-approximated in $\widetilde{O}(mn^{1/3}\varepsilon^{-11/3})$ time.*

3.4.5 Approximating the Value of the Maximum s-t Flow in Time $\widetilde{O}(m + n^{4/3}\varepsilon^{-8/3})$

By applying our improved algorithm described by Theorem 3.4.3 to a cut sparsifier of G (cf. Corollary 2.6.2) gives us an algorithm for $(1-\varepsilon)$-approximating the value of the maximum s-t flow on G in time $\widetilde{O}(m + n^{4/3}\varepsilon^{-3})$.

We note that this only allows us to approximate the *value* of the maximum s-t flow on G. It gives us a flow on G', not one on G. We do not know how to use an approximately maximum s-t flow on G' to obtain one on G in less time than would be required to compute a maximum flow in G from scratch using the algorithm from Section 3.4.

For this reason, there is a gap between the time we require to find a maximum flow and the time we require to compute its value. We note, however, that this gap will not exist for the minimum s-t cut problem, since an approximately minimum s-t cut on G' will also be an approximately minimum s-t cut G. We will present an algorithm for finding such a cut in the next section. By the Max Flow-Min Cut Theorem, this will provide us with an alternate algorithm for approximating the value of the maximum s-t flow. It will have a slightly better dependence on ε, which will allow us to approximate the value of the maximum s-t flow in time $\widetilde{O}(m + n^{4/3}\varepsilon^{-8/3})$.

3.5 A Dual Algorithm for Finding an Approximately Minimum s-t Cut in Time $\widetilde{O}(m + n^{4/3}\varepsilon^{-8/3})$

In this section, we'll describe our approach from a dual perspective that yields an even simpler algorithm for computing an approximately minimum s-t cut. Rather than using electrical flows to obtain a flow, it will use the electrical potentials to obtain a cut.

The algorithm will eschew the oracle abstraction and multiplicative-weights-update machinery discussed in Section 2.8. Instead, it will just repeatedly compute an electrical flow, increase the resistances of edges according to the amount flowing over them, and repeat. It will then use the electrical potentials of the last flow computed to find a cut by picking a cutoff and splitting the vertices according to whether their potentials are above or below the cutoff.

The algorithm is shown in Figure 3-4. It finds a $(1+\varepsilon)$-approximately minimum s-t cut in time $\widetilde{O}(m^{4/3}\varepsilon^{-8/3})$; applying it to a cut sparsifier will give us

Theorem 3.5.1. *For any $0 < \varepsilon < 1/7$, we can find a $(1+\varepsilon)$-approximate minimum s-t cut in $\widetilde{O}(m + n^{4/3}\varepsilon^{-8/3})$ time.*

We note that, in this algorithm, there is no need to deal explicitly with edges flowing more than ρ, maintain a set of forbidden edges, or average the flows from different steps. We will separately study edges with very large flow in our analysis, but the algorithm itself avoids the complexities that appeared in the improved flow algorithm described in Section 3.4.

We further note that the update rule is slightly modified from the one that appeared earlier in this chapter. This is done to guarantee that the effective resistance increases substantially when some edge flows more than ρ, without having to explicitly cut it. Our previous rule allowed the weight (but not resistance) of an edge to constitute a very small fraction of the total weight; in this case, a significant multiplicative increase in the weight of an edge may not produce a substantial change in the effective resistance of the graph.

3.5.1 An Overview of the Analysis

To analyze this algorithm, we will track the total weight placed on the edges crossing some minimum s-t cut. The basic observation for our analysis is that the same amount of net flow must be sent across every cut, so edges in small cuts will tend to have higher congestion than edges in large cuts. Since our algorithm increases the weight of an edge according to its congestion, this will cause our algorithm to concentrate a larger and larger fraction of the total weight on the small cuts of the graph. This will continue until almost all of the weight is concentrated on approximately minimum cuts.

Of course, the edges crossing a minimum s-t cut will also cross many other (likely much larger) cuts, so we certainly can't hope to obtain a graph in which no edge

Input : A graph $G = (V, E)$ with capacities $\{u_e\}_e$, and a target flow value F
Output: A cut C

Initialize $w_e^0 \leftarrow 1$ for all edges e, $\rho \leftarrow 3m^{1/3}\varepsilon^{-2/3}$, $N \leftarrow 5\varepsilon^{-8/3}m^{1/3}\ln m$, and $\delta \leftarrow \varepsilon^2$.
for $i := 1, \ldots, N$ **do**
  Find an approximate electrical flow $\tilde{f}^{i-1}$ and potentials $\tilde{\phi}$ using Theorem 3.3.1 on G with
    resistances $r_e^{i-1} = \frac{w_e^{i-1}}{u_e^2}$, target flow value F, and parameter δ.
  $\mu^{i-1} \leftarrow \sum_e w_e^{i-1}$
  $w_e^i \leftarrow w_e^{i-1} + \frac{\varepsilon}{\rho}\mathsf{cong}_{\tilde{f}^{i-1}}(e)w_e^{i-1} + \frac{\varepsilon^2}{m\rho}\mu^{i-1}$ for each $e \in E$
  Scale and translate $\tilde{\phi}$ so that $\widetilde{\phi}_s = 1$ and $\widetilde{\phi}_t = 0$
  Let $C_x = \{v \in V \mid \phi_v > x\}$
  Set C to be the set C_x of minimal capacity
  If the capacity of C_x is less than $F/(1-7\varepsilon)$, **return** C_x.
end
return "fail"

Figure 3-4: A dual algorithm for finding an s-t cut

crossing a large cut has non-negligible weight. In order to formalize the above argument, we will thus need some good way to measure the extent to which the weight is "concentrated on approximately minimum s-t cuts".

In Section 3.5.2, we will show how to use effective resistance to formulate such a notion. In particular, we will show that if we can make the effective resistance large enough then we can find a cut of small capacity. In Section 3.5.3, we will use an argument like the one described above to show that the resistances produced by the algorithm in Figure 3-4 converge after $T = \widetilde{O}(m^{1/3}\varepsilon^{-8/3})$ steps to one that meets such a bound.

3.5.2 Cuts, Electrical Potentials, and Effective Resistance

During the algorithm, we scale and translate the potentials of the approximate electrical flow so that $\widetilde{\phi}_s = 1$ and $\widetilde{\phi}_t = 0$. We then produce a cut by choosing $x \in [0, 1]$ and dividing the graph into the sets $C = \{v \in V \mid \phi_v > x\}$ and $V \setminus C = \{v \in V \mid \widetilde{\phi}_v \leq x\}$. The following lemma upper bounds the capacity of the resulting cut in terms of the electrical potentials and edge capacities.

Lemma 3.5.2. *Let $\tilde{\phi}$ be as above. Then there is a cut C_x of capacity at most*

$$\sum_{(u,v)\in E} |\widetilde{\phi}_u - \widetilde{\phi}_v| u_{(u,v)}. \tag{3.12}$$

Proof. Consider choosing $x \in [0, 1]$ uniformly at random. The probability that an edge (u, v) is cut is precisely $|\widetilde{\phi}_u - \widetilde{\phi}_v|$. So, the expected capacity of the edges in a random cut is given by (3.12), and so there is a cut of capacity at most (3.12). □

Now, suppose that one has a fixed total amount of resistance μ to distribute over the edges of a cut of size F. It is not difficult to see that the maximum possible effective resistance between s and t in such a case is $\frac{\mu}{F^2}$, and that this is achieved when one puts a resistance of $\frac{\mu}{F}$ on each of the edges. This suggests the following lemma, which bounds the quantity in Lemma 3.5.2 in terms of the effective resistance and the total resistance (appropriately weighted when the edges have non-unit capacities):

Lemma 3.5.3. *Let $\mu = \sum_e u_e^2 r_e$, and let the effective s-t resistance of G with edge resistances given by $\boldsymbol{r}$ be $R_{\text{eff}}(\boldsymbol{r})$. Let $\boldsymbol{\phi}$ be the potentials of the electrical s-t flow, scaled to have potential drop 1 between s and t. Then*

$$\sum_{(u,v)\in E} |\phi_u - \phi_v| u_{(u,v)} \leq \sqrt{\frac{\mu}{R_{\text{eff}}(\boldsymbol{r})}}.$$

If, $\tilde{\boldsymbol{\phi}}$ is an approximate electrical potential returned by the algorithm of Theorem 3.3.1 when run with parameter $\delta \leq 1/3$, re-scaled to have potential difference 1 between s and t, then

$$\sum_{(u,v)\in E} |\widetilde{\phi}_u - \widetilde{\phi}_v| u_{(u,v)} \leq (1+2\delta)\sqrt{\frac{\mu}{R_{\text{eff}}(\boldsymbol{r})}}.$$

Proof. By Fact 2.4.1, the rescaled true electrical potentials correspond to a flow of value $1/R_{\text{eff}}(\boldsymbol{r})$ and

$$\sum_{(u,v)\in E} \frac{(\phi_u - \phi_v)^2}{r_{(u,v)}} = \frac{1}{R_{\text{eff}}(\boldsymbol{r})}.$$

So, we can apply the Cauchy-Schwarz inequality to prove

$$\begin{aligned}\sum_{(u,v)\in E} |\phi_u - \phi_v| u_{(u,v)} &\leq \sqrt{\sum_{(u,v)\in E} \frac{(\phi_u - \phi_v)^2}{r_{(u,v)}} \sum_e u_e^2 r_e} \\ &= \sqrt{\frac{\mu}{R_{\text{eff}}(\boldsymbol{r})}}.\end{aligned}$$

By parts a and c of Theorem 3.3.1, after we rescale $\tilde{\boldsymbol{\phi}}$ to have potential drop 1 between s and t, it will have energy

$$\sum_{(u,v)\in E} \frac{(\widetilde{\phi}_u - \widetilde{\phi}_v)^2}{r_e} \leq \frac{1+\delta}{1-\delta}\frac{1}{R_{\text{eff}}(\boldsymbol{r})} \leq (1+3\delta)\frac{1}{R_{\text{eff}}(\boldsymbol{r})},$$

for $\delta \leq 1/3$. The rest of the analysis follows from another application of Cauchy-Schwarz. □

3.5.3 The Proof that the Dual Algorithm Finds an Approximately Minimum Cut

We'll show that if $F \geq F^*$ then within $T = 5\varepsilon^{-8/3}m^{1/3}\ln m$ iterations, the algorithm in Figure 3-4 will produce a set of resistances $\boldsymbol{r}^i$ such that

$$R_{\text{eff}}(\boldsymbol{r}^i) \geq (1-7\varepsilon)\frac{\mu^i}{(F)^2}. \tag{3.13}$$

Once such a set of resistances has been obtained, Lemmas 3.5.2 and 3.5.3 tell us that the best potential cut of $\tilde{\boldsymbol{\phi}}$ will have capacity at most

$$\frac{1+2\delta}{\sqrt{1-7\varepsilon}}F \leq \frac{1}{1-7\varepsilon}F.$$

The algorithm will then return this cut.

Let C be the set of edges crossing some minimum cut in our graph. Let $u_C = F^*$ denote the capacity of the edges in C. We will keep track of two quantities: the weighted geometric mean of the weights of the edges in C,

$$\nu^i = \left(\prod_{e\in C}\left(w_e^i\right)^{u_e}\right)^{1/u_C},$$

and the total weight

$$\mu^i = \sum_e w_e^i = \sum_e r_e^i u_e^2$$

of the edges of the entire graph. Clearly $\nu^i \leq \max_{e\in C} w_e^i$. In particular,

$$\nu^i \leq \mu^i$$

for all i.

Our proof that the effective resistance cannot remain large for too many iterations will be similar to our analysis of the flow algorithm in Section 3.4. We suppose to the contrary that $R_{\text{eff}}(\boldsymbol{r}^i) \leq (1-7\varepsilon)\frac{\mu^i}{(F)^2}$ for each $1 \leq i \leq T$. We will show that, under this assumption:

1. The total weight μ^i doesn't get too large over the course of the algorithm **[Lemma 3.5.4]**.

2. The quantity ν^i increases significantly in any iteration in which no edge has congestion more than ρ **[Lemma 3.5.5]**. Since μ^i doesn't get too large, and $\nu^i \leq \mu^i$, this will not happen too many times.

3. The effective resistance increases significantly in any iteration in which some edge has congestion more than ρ **[Lemma 3.5.6]**. Since μ^i does not get too large, and the effective resistance is assumed to be bounded in terms of the total weight μ^i, this cannot happen too many times.

The combined bounds from 2 and 3 will be less than T, which will yield a contradiction.

Lemma 3.5.4. *For each $i \leq T$ such that $R_{\text{eff}}^{\leq}(\boldsymbol{r}^i)(1-7\varepsilon)\frac{\mu^i}{F^2}$,*

$$\mu^{i+1} \leq \mu^i \exp\left(\frac{\varepsilon(1-2\varepsilon)}{\rho}\right).$$

So, if for all $i \leq T$ we have $R_{\text{eff}}^{\leq}(\boldsymbol{r}^i)(1-7\varepsilon)\frac{\mu^i}{F^2}$, then

$$\mu^T \leq \mu^0 \exp\left(\frac{\varepsilon(1-2\varepsilon)}{\rho}T\right). \tag{3.14}$$

Proof. If $R_{\text{eff}}^{\leq}(\boldsymbol{r}^i)(1-7\varepsilon)\frac{\mu^i}{F^2}$ then the electrical flow f of value F has energy

$$F^2 R_{\text{eff}}^{=}(\boldsymbol{r}^i)\sum \mathsf{cong}_{f^i}(e)^2 w_e^i \leq (1-7\varepsilon)\mu^i.$$

By Theorem 3.3.1, the approximate electrical flow $\tilde{f}$ has energy at most $(1+\delta)$ times the energy of f,

$$\sum \mathsf{cong}_{\tilde{f}^i}(e)^2 w_e^i \leq (1+\delta)(1-7\varepsilon)\mu^i \leq (1-6\varepsilon)\mu^i.$$

Applying the Cauchy-Schwarz inequality, we find

$$\sum \mathsf{cong}_{\tilde{f}^i}(e) w_e^i \leq \sqrt{\sum w_e^i}\sqrt{\sum \mathsf{cong}_{\tilde{f}^i}(e)^2 w_e^i} \leq \sqrt{1-6\varepsilon}\mu^i \leq (1-3\varepsilon)\mu^i.$$

Now we can compute

$$\begin{aligned}
\mu^{i+1} &= \sum_e w_e^{i+1} \\
&= \sum_e w_e^i + \frac{\varepsilon}{\rho}\sum_e w_e^i \mathsf{cong}_{f^i}(e) + \frac{\varepsilon^2}{\rho}\mu^i \\
&\leq \left(1 + \frac{\varepsilon(1-2\varepsilon)}{\rho}\right)\mu^i \\
&\leq \mu^i \exp\left(\frac{\varepsilon(1-2\varepsilon)}{\rho}\right)
\end{aligned}$$

as desired. □

Lemma 3.5.5. *If $\mathsf{cong}_{f^i}(e) \leq \rho$ for all e, then*

$$\nu^{i+1} \geq \exp\left(\frac{\varepsilon(1-\varepsilon)}{\rho}\right)\nu^i.$$

Proof. To bound the increase of ν^{i+1} over ν^i, we use the inequality

$$(1+\varepsilon x) \geq \exp\left(\varepsilon(1-\varepsilon)x\right),$$

which holds for ε and x between 0 and 1. We apply this inequality with $x = \mathsf{cong}_{\tilde{f}^i}(e)/\rho$. As $\tilde{f}^i$ is a flow of value F and C is a cut, $\sum_{e\in C}\left|\tilde{f}_e^i\right| \geq F$. We now compute

$$\begin{aligned}
\nu^{i+1} &= \left(\prod_{e\in C}\left(w_e^{t+1}\right)^{u_e}\right)^{1/u_C} \\
&\geq \left(\prod_{e\in C}\left(w_e^i\left(1+\frac{\varepsilon}{\rho}\mathsf{cong}_{\tilde{f}^i}(e)\right)\right)^{u_e}\right)^{1/u_C} \\
&= \nu^i\left(\prod_{e\in C}\left(1+\frac{\varepsilon}{\rho}\mathsf{cong}_{\tilde{f}^i}(e)\right)^{u_e}\right)^{1/u_C} \\
&\geq \nu^i \exp\left(\frac{1}{u_C}\sum_{e\in C} u_e\frac{\varepsilon(1-\varepsilon)}{\rho}\mathsf{cong}_{\tilde{f}^i}(e)\right) \\
&= \nu^i \exp\left(\frac{1}{u_C}\sum_{e\in C}\frac{\varepsilon(1-\varepsilon)}{\rho}\left|\tilde{f}_e^i\right|\right) \\
&\geq \nu^i \exp\left(\frac{1}{u_C}\frac{\varepsilon(1-\varepsilon)}{\rho}F\right) \\
&\geq \nu^i \exp\left(\frac{\varepsilon(1-\varepsilon)}{\rho}\right).
\end{aligned}$$

□

Lemma 3.5.6. *If* $R_{\text{eff}}(\boldsymbol{r}^i) \leq (1-7\varepsilon)\frac{\mu^i}{F^2}$ *and there exists some edge* e *such that* $\mathsf{cong}_{\tilde{f}^i}(e) > \rho$*, then*

$$R_{\text{eff}}(\boldsymbol{r}^{i+1}) \geq \exp\left(\frac{\varepsilon^2\rho^2}{4m}\right) R_{\text{eff}}(\boldsymbol{r}^i).$$

Proof. We first show by induction that

$$w_e^i \geq \frac{\varepsilon}{m}\mu^i.$$

If $i = 0$ we have $1 \geq \varepsilon$. For $i > 0$, we have

$$\begin{aligned} w_e^{i+1} &\geq w_e^i + \frac{\varepsilon^2}{m\rho}\mu^i \\ &\geq \left(\frac{\varepsilon}{m} + \frac{\varepsilon^2}{m\rho}\right)\mu^i \\ &= \frac{\varepsilon}{m}\left(1 + \frac{\varepsilon}{\rho}\right)\mu^i \\ &\geq \frac{\varepsilon}{m}\exp\left(\frac{\varepsilon(1-2\varepsilon)}{\rho}\right)\mu^i \\ &\geq \frac{\varepsilon}{m}\mu^{i+1}, \end{aligned}$$

by Lemma 3.5.4.

We now show that an edge e for which $\mathsf{cong}_{\tilde{f}^i}(e) \geq \rho$ contributes a large fraction of the energy to the true electrical flow of value F. By the assumptions of the lemma, the energy of the true electrical flow of value F is

$$F^2 R_{\text{eff}}(\boldsymbol{r}) \leq (1 - 7\varepsilon)\mu^i.$$

On the other hand, the energy of the edge e in the approximate electrical flow is

$$\tilde{f}^i(e)^2 r_e^i = \mathsf{cong}_{\tilde{f}^i}(e)^2 w_e \geq \rho^2 \frac{\varepsilon}{m}\mu^i.$$

As a fraction of the energy of the true electrical flow, this is at least

$$\frac{1}{1-7\varepsilon}\frac{\rho^2\varepsilon}{m} = \frac{1}{(1-7\varepsilon)\varepsilon^{1/3}m^{1/3}}.$$

By part b of Theorem 3.3.1, the fraction of the energy that e accounts for in the true flow is at least

$$\frac{1}{(1-7\varepsilon)\varepsilon^{1/3}m^{1/3}} - \frac{\varepsilon^2}{2mR} \geq \frac{1}{\varepsilon^{1/3}m^{1/3}} = \frac{\rho^2\varepsilon}{m}.$$

As

$$w_e^{i+1} \geq w_e^i\left(1 + \frac{\varepsilon}{\rho}\mathsf{cong}_{\tilde{f}^i}(e)\right) \geq (1+\varepsilon)w_e^i,$$

we have increased the weight of edge e by a factor of at least $(1 + \varepsilon)$. So, by Lemma 3.4.4,

$$\frac{R_{\text{eff}}(\boldsymbol{r}^{i+1})}{R_{\text{eff}}(\boldsymbol{r}^i)} \geq \left(1 + \frac{\rho^2\varepsilon^2}{2m}\right) \geq \exp\left(\frac{\rho^2\varepsilon^2}{4m}\right).$$

□

We now combine these lemmas to obtain our main bound:

Lemma 3.5.7. *For $\varepsilon \leq 1/7$, after T iterations, the algorithm in Figure 3-4 will produce a set of resistances such that $R_{\text{eff}}^{\leq}(\boldsymbol{r}^i)(1-7\varepsilon)\frac{\mu^i}{F^2}$.*

Before proving Lemma 3.5.7, we note that combining it with Lemmas 3.5.2 and 3.5.3 immediately implies our main result:

Theorem 3.5.8. *On input $\varepsilon < 1/7$, the algorithm in Figure 3-4 runs in time $\widetilde{O}(m^{4/3}\varepsilon^{-8/3})$. If $F \geq F^*$, then it returns a cut of capacity at most $F/(1-7\varepsilon)$, where F^* is the minimum capacity of an s-t cut.*

To use this algorithm to find a cut of approximately minimum capacity, one should begin as described in Section 3.3 by crudely approximating the minimum cut, and then applying the above algorithm in a binary search. Crudely analyzed, this incurs a multiplicative overhead of $O(\log m/\varepsilon)$.

Proof of Lemma 3.5.7. Suppose as before that $R_{\text{eff}}(\boldsymbol{r}^i) \leq (1-7\varepsilon)\frac{\mu^i}{F^2}$ for all $1 \leq i \leq T$. By Lemma 3.5.4 and the fact that $\mu^0 = m$, the total weight at the end of the algorithm is bounded by

$$\mu^T \leq \mu^0 \exp\left(\frac{\varepsilon(1-2\varepsilon)}{\rho}T\right) = m\exp\left(\frac{\varepsilon(1-2\varepsilon)}{\rho}T\right). \tag{3.15}$$

Let $A \subseteq \{1,\ldots,T\}$ be the set of i for which $\mathsf{cong}_{\tilde{f}^i}(e) \leq \rho$ for all e, and let $B \subseteq \{1,\ldots,T\}$ be the set of i for which there exists an edge e with $\mathsf{cong}_{\tilde{f}^i}(e) > \rho$. Let $a = |A|$ and $b = |B|$, and note that $a + b = T$. We will obtain a contradiction by proving bounds on a and b that add up to something less than T.

We begin by bounding a. By applying Lemma 3.5.5 to all of the steps in A and noting that ν^i never decreases during the other steps, we obtain

$$\nu^T \geq \exp\left(\frac{\varepsilon(1-\varepsilon)}{\rho}a\right)\nu^0 = \exp\left(\frac{\varepsilon(1-\varepsilon)}{\rho}a\right) \tag{3.16}$$

Since $\nu^T \leq \mu^T$, we can combine equations (3.15) and (3.16) to obtain

$$\exp\left(\frac{\varepsilon(1-\varepsilon)}{\rho}a\right) \leq m\exp\left(\frac{\varepsilon(1-2\varepsilon)}{\rho}T\right).$$

Taking logs of both sides and solving for a yields

$$\frac{\varepsilon(1-\varepsilon)}{\rho}a \leq \frac{\varepsilon(1-2\varepsilon)}{\rho}T + \ln m,$$

from which we obtain

$$a \leq \frac{1-2\varepsilon}{1-\varepsilon}T + \frac{\rho}{\varepsilon(1-\varepsilon)}\ln m \leq (1-\varepsilon)T + \frac{\rho}{\varepsilon(1-\varepsilon)}\ln m < (1-\varepsilon)T + \frac{7\rho}{6\varepsilon}\ln m. \tag{3.17}$$

We now use a similar argument to bound b. By applying Lemma 3.5.6 to all of the steps in B and noting that $R_{\text{eff}}(\boldsymbol{r}^i)$ never decreases during the other steps, we

obtain

$$R_{\text{eff}}(\boldsymbol{r}^T) \geq \exp\left(\frac{\varepsilon^2\rho^2}{4m}b\right) R_{\text{eff}}(\boldsymbol{r}^0) \geq \exp\left(\frac{\varepsilon^2\rho^2}{4m}b\right)\frac{1+\varepsilon}{m^2F^{*2}} \geq \exp\left(\frac{\varepsilon^2\rho^2}{4m}b\right)\frac{1+\varepsilon}{m^2F^2},$$

where the second inequality applies the bound $R_{\text{eff}}(\boldsymbol{r}^0) \geq 1/(m^2F^{*2})$, which follows from an argument like the one used in equation (3.9). Using our assumption that $R_{\text{eff}}(\boldsymbol{r}^T) \leq (1-7\varepsilon)\frac{\mu^T}{(F)^2}$ and the bound on μ^T from equation (3.15), this gives us

$$\exp\left(\frac{\varepsilon^2\rho^2}{4m}b\right)\frac{1+\varepsilon}{m^2F^2} \leq \frac{1-7\varepsilon}{F^2}m\exp\left(\frac{\varepsilon(1-2\varepsilon)}{\rho}T\right).$$

Taking logs and rearranging terms gives us

$$b \leq \frac{4m(1-2\varepsilon)}{\varepsilon\rho^3}T + \frac{4m}{\varepsilon^2\rho^2}\ln\left(\frac{1-7\varepsilon}{1+\varepsilon}m^3\right) < \frac{4m}{\varepsilon\rho^3}T + \frac{12m}{\varepsilon^2\rho^2}\ln m. \tag{3.18}$$

Adding the inequalities in equations (3.17) and (3.18), grouping terms, and plugging in the values of ρ and T, yields

$$\begin{aligned}
T = a + b &< \left((1-\varepsilon) + \frac{4m}{\varepsilon\rho^3}\right)T + \left(\frac{7\rho}{6\varepsilon} + \frac{12m}{\varepsilon^2\rho^2}\right)\ln m \\
&= \left((1-\varepsilon) + \frac{4m}{\varepsilon(27m\varepsilon^{-2})}\right)(5\varepsilon^{-8/3}m^{1/3}\ln m) \\
&+ \left(\frac{7m^{1/3}\varepsilon^{-2/3}}{2\varepsilon} + \frac{12m}{\varepsilon^2(9m^{2/3}\varepsilon^{-4/3})}\right)\ln m \\
&= \left((1-\varepsilon) + \frac{4\varepsilon}{27}\right)(5\varepsilon^{-8/3}m^{1/3}\ln m) + \left(\frac{7m^{1/3}}{2\varepsilon^{5/3}} + \frac{12m^{1/3}}{9\varepsilon^{2/3}}\right)\ln m \\
&= \frac{5m^{1/3}}{\varepsilon^{8/3}}\ln m - \left(\frac{41}{54} - \frac{12\varepsilon}{9}\right)\frac{m^{1/3}}{\varepsilon^{5/3}}\ln m \\
&< \frac{5m^{1/3}}{\varepsilon^{8/3}} = T.
\end{aligned}$$

This is our desired contradiction, which completes the proof. □

3.6 Towards a Maximum Flow Algorithm for Directed Graphs

The whole discussion in this chapter was focused on dealing with undirected graphs. So, given that the techniques we employed here seem to be applicable only to undirected setting, at first one might not expect our approach to be also relevant to directed case of the maximum s-t flow problem.

Somewhat surprisingly, however, it turns out that there is a quite close connection between the maximum s-t flow problem in directed and undirected graphs. Namely,

one can turn an efficient algorithm that computes the maximum s-t flow in undirected graphs into an efficient algorithm that computes such a flow also in directed graphs. More precisely, the following theorem holds.

Theorem 3.6.1 (Folklore)**.** *Given an algorithm A that computes a maximum s-t flow in any* undirected *capacitated graph $G' = (V', E', \boldsymbol{u}')$ in time $T(|V'|, |E'|, U')$, where $U' := \max_{e,e'} \frac{u'_e}{u'_{e'}}$ is the capacity ratio of G', one can compute a maximum s-t flow in any* directed *capacitated graph $G = (V, E, \boldsymbol{u})$ in time $T(O(|V|), O(|E|), |V| \cdot U) + \widetilde{O}(|E|)$, where $U := \max_{e,e'} \frac{u_e}{u_{e'}}$ is the capacity ratio of G.*

Note that in the above theorem we need the algorithm A to compute the exact maximum s-t flow not only its approximation. So, we cannot directly use here the algorithm presented in this chapter. Still, however, this theorem shows that to make progress also on the directed case, we can just focus on refining our approach to develop an error control mechanism that allows us to perform exact undirected maximum flow computations. We also observe that the maximum s-t flow and minimum s-t cut duality [67, 57] can be used to get an analogous result for the minimum s-t cut problem.

Proof. The way we will find the maximum s-t flow f^* in the graph G within desired time bounds – and thus prove the theorem – is based on constructing an appropriate undirected graph $\widetilde{G} = (\widetilde{V}, \widetilde{E}, \widetilde{\boldsymbol{u}})$ and extracting the flow f^* from the maximum s-t flow $\tilde{f}^*$ in $\widetilde{G}$ (that we will find using the algorithm A).

The graph $\widetilde{G}$ is constructed from G as follows. We take $\widetilde{V}$ to be equal to V. Next, for each (directed) arc $e = (u, v)$ of G we add (undirected) edges $h(e) := (u, v)$, $h^s(e) := (s, v)$, and $h^t(e) := (u, t)$ to $\widetilde{G}$, with the capacity of each of these edges being u_e. Clearly, the resulting graph $\widetilde{G}$ is undirected and has $O(|E|)$ edges and $O(|V|)$ vertices. Also, by just viewing each multi-edge created in $\widetilde{G}$ as a singe edge with total capacity being the sum of the capacities of all the edges forming it, we see that the capacity ratio of $\widetilde{G}$ is at most $|V| \cdot U$.

Now, we claim that the maximum s-t flow value $\widetilde{F}^*$ of the graph $\widetilde{G}$ is at least

$$\widetilde{F}^* \geq 2F^* + \sum_{e \in E} u_e,$$

where F^* is the value of the maximum s-t flow in G.

To see why this is the case, we just need to realize that each s-t cut C in $\widetilde{G}$ has a capacity of at least $2F^* + \sum_{e \in E} u_e$. The claim will then follow from the Max-Flow Min-Cut theorem [67, 57]. To this end, let us fix some s-t cut C. Next, let us consider an arc $e \in E$ that is leaving this cut in G. By construction of $\widetilde{G}$, all the edges $h(e)$, $h^s(e)$, and $h^t(e)$ contribute to the capacity of the cut C in $\widetilde{G}$ and their total contribution is exactly $3u_e$. On the other hand, for every arc $e \in E$ that is not leaving C, we have that either $h(e)$, $h^s(e)$, or $h^t(e)$ has to be cut by C (as these three edges form an s-t path), which results in a contribution of at least u_e to the capacity of C in $\widetilde{G}$. Now, the claim follows by invoking Max-Flow Min-Cut theorem again to

show that the total capacity $u(C)$ of the arcs leaving the cut C in G has to be at least F^*.

In the light of the above, if we compute the maximum s-t flow $\tilde{f}^*$ in $\widetilde{G}$ using the algorithm A, then the running time of this procedure will be $T(O(|V|), O(|E|), |V| \cdot U) + O(|E|)$ and the value $|\tilde{f}^*|$ of $\tilde{f}^*$ is at least $2F^* + \sum_{e \in E} u_e$.

Now, we describe how to extract out of the flow $\tilde{f}^*$ a feasible flow f^* in G of value at least F^* in $\widetilde{O}(|E|)$ time. Clearly, once we obtain this flow f^* it will be the maximum s-t flow in G that we are looking for. The extraction will consists of two steps.

The first step is based on first obtaining a flow $\tilde{f}'$ out of $\tilde{f}^*$ by subtracting for each arc $e \in E$, the s-t flows f^e in G that sends u_e units of flow through the arcs $h^s(e)$, $h(e)$, and $h^t(e)$; and then scaling it down by a factor of 2. In other words, we take

$$\tilde{f}' := \frac{1}{2}\left(\tilde{f}^* - \sum_{e \in E} f^e\right).$$

Note that the value $|\tilde{f}'|$ of this flow is at least $\frac{1}{2}(\widetilde{F}^* - \sum_{e \in E} u_e) \geq F^*$. Furthermore, it is easy to see that by the feasibility of $\tilde{f}$, $\tilde{f}'$ routes over each edge $h(e)$ at most $\frac{1}{2}(|\tilde{f}(h(e))| + f^e(h(e))) \leq u_e$ units of flow and this flow has to flow in direction consistent with the orientation of the arc e. So, if $\tilde{f}'$ would not flow any flow over the edges $h^s(e)$s and $h^t(e)$s, i.e., we had $|\tilde{f}'(h^s(e))| = |\tilde{f}'(h^t(e))| = 0$, for each $e \in E$, then $\tilde{f}'$ would be also a feasible flow in G and thus we could just take f^* to be $\tilde{f}'$.

Of course, we cannot expect the flow $\tilde{f}'$ to never flow anything over the edges $h^s(e)$ and $h^t(e)$. However, what is still true is that the only direction the flow over the edge $h^s(e)$ (resp. the edge $h^t(e)$) can flow is towards s (resp. out of t). This is so, since by feasibility of $\tilde{f}$, the flow over the edges $h^s(e)$ and $h^t(e)$ in $\tilde{f}$ can be at most u_e, and during construction of $\tilde{f}'$ we subtract the s-t flows f^es that flow exactly u_e units over the edges $h^s(e)$ (resp. $h^t(e)$) in the direction out of s (resp. towards t).

However, as an acyclic s-t flow does not send any flow towards s or out of t, it must be the case that the flow $\tilde{f}'$ can have a non-zero flow on edges $h^s(e)$ and $h^t(e)$ only if it contains some flow cycles. Therefore, if we just apply to $\tilde{f}'$ the $\widetilde{O}(m)$-time flow-cycle-canceling algorithm that is described, e.g., in [74], then the acyclic flow f^* that we obtain in this way will be the maximum s-t flow in G that we are seeking. □

Chapter 4

Multicommodity Flow Problems

We turn our attention to multicommodity flow problems – natural generalizations of the maximum s-t flow problem we considered in the previous chapter.[1] We investigate the task of designing $(1-\varepsilon)$-approximation algorithms for such problems and develop a new method of speeding up the existing algorithms for them. This method is based on employing the ideas from an area of dynamic graph algorithms to improve the performance of the multiplicative-weights-update-based algorithms that are traditionally used in the context of multicommodity flows. As a result, we achieve running time improvements by a factor of roughly $\widetilde{O}(m/n)$. Also, whenever ε is fixed, our running times essentially match a natural barrier for all the existing fast algorithms and are within a $\widetilde{O}(m/n)$ factor of an absolute lower bound.

4.1 Introduction

We will be studying multicommodity flow problems. These problems correspond to a task of finding in a capacitated and directed graph $G = (V, E, \boldsymbol{u})$, a set of k flows $(f_1, \ldots, f_k)$ – each f_i being an s_i-t_i flow of commodity i with a source s_i and a sink t_i – that optimize some objective function while respecting the capacity constraints, i.e., while having the total flow $\sum_i f_i(e)$ of the commodities through any arc e not exceeding its capacity u_e. There are two basic variations of this task. The first one is the *maximum multicommodity flow* problem – in this case the objective is to maximize the sum $\sum_i |f_i|$ of the values of the flows. The second one is the *maximum concurrent flow* problem. In this problem, we are given a set of k positive *demands* $d_1, \ldots, d_k$ and are asked to find a multicommodity flow that is feasible (i.e. obeys arc capacities) and routes θd_i units of commodity i between each source-sink pair (s_i, t_i). The goal is to maximize the value of the *rate* θ.

Although the problems defined above can be solved optimally in polynomial time by formulating them as linear programs, in many applications it is more important to compute an approximate solution fast than to compute an optimal one. Therefore, much effort was put into obtaining efficient approximation algorithms for these problems. In particular, one is interested in getting a fast algorithm that computes a

[1] This chapter contains material from [106].

solution that has objective value within a factor of $(1-\varepsilon)$ of the optimal one, where $\varepsilon > 0$ is the desired accuracy factor.

4.1.1 Previous Work

Over the past two decades there has been a rich history of results providing such a $(1-\varepsilon)$-approximation algorithms for multicommodity flow problems. The dominating approach stemmed from Lagrangian relaxation methods and can be viewed as a precursor of the multiplicative-weight-update-based approach. Shahrokhi and Matula [122] presented the first such algorithm for the maximum concurrent flow problem with uniform arc capacities that was combinatorial and introduced the idea of using an exponential length function to control arc congestion. Building on that, a series of results [94, 100, 72, 76, 113, 124, 116, 85, 77] based on Langrangian relaxation and linear programming decomposition yielded algorithms that had significantly improved running time and could be applied to various versions of the multicommodity flow problem with arbitrary arc capacities. All the above algorithms were based on computing an initial (infeasible) flow and then redistributing it from more congested paths to less congested ones by repeatedly solving an oracle subproblem of either minimum cost single-commodity flow [100, 72, 76, 116, 85, 77], or shortest path [122, 94, 113, 124].

In [139], Young deviated from this theme by presenting an *oblivious rounding* algorithm that avoids rerouting of the flow. Instead, it builds the solution from scratch. At each step it employs shortest path computations (with respect to exponential length function that models the congestion of the arcs) to augment the flow along suitable (i.e. relatively uncongested) paths. At the end, it obtains the final feasible solution by scaling down the flow by the maximum congestion it incurred on arcs. A similar approach was taken by Garg and Könemann [70]; however they managed to provide an elegant framework for solving multicommodity flow problems that yields a simple analysis of the correctness of the obtained algorithms. This allowed them to match and, in some cases, improve over the running time of the algorithms obtained via the redistribution methodology. Subsequently, Fleischer [65] used this framework to develop significantly faster algorithms for various multicommodity flow problems. In particular, for the maximum multicommodity flow problem she managed to obtain a running time of $\widetilde{O}(m^2\varepsilon^{-2})$ that is independent of the number of commodities. For the maximum concurrent flow problem her algorithm has a running time of $\widetilde{O}((m+k)m\varepsilon^{-2})$. Her results for both versions of the concurrent flow problem were later improved by Karakostas [84], who was able to reduce the term in the running time that depends on k from $\widetilde{O}(km\varepsilon^{-2})$ to $\widetilde{O}(kn\varepsilon^{-2})$. Interestingly, he also showed that if we want to obtain the $(1-\varepsilon)$-approximation of only the *value* of the maximum concurrent flow rate (without obtaining the actual flow) then this can be done in $\widetilde{O}(m^2\varepsilon^{-2})$ time.

All the above algorithms have quadratic dependence of their running times on $1/\varepsilon$. Klein and Young [95] gave an evidence that this quadratic dependence might

be inherent for *Dantzig-Wolfe-type* algorithms[2] and all the algorithms mentioned so far are of this type. As it turns out, better dependence on $1/\varepsilon$ can be obtained. Bienstock and Iyengar [29] adapted the technique of Nesterov [108] to give an $(1-\varepsilon)$-approximation algorithm for maximum concurrent flow problem that have $O(\frac{1}{\varepsilon \log 1/\varepsilon})$ dependence on $1/\varepsilon$. Very recently, Nesterov [109] obtained an algorithm for this problem in which this dependence is just linear. However, the running times of both these algorithms have worse dependence on parameters other than $1/\varepsilon$ compared to the algorithms described above – for example, the approximation scheme due to Nesterov has the running time of $\widetilde{O}(k^2m^2\varepsilon^{-1})$.

4.1.2 Overview of Our Results

The point of start of the results to be presented in this chapter are the multiplicative-weights-update-based algorithms of Garg and Könemann [70] and of Fleischer [65] for multicommodity flow problems. As mentioned above, at a high level, these algorithms reduce the task of approximation of multicommodity flow problems to solving a sequence of shortest-path questions in the underlying graph with the lengths of edges modeling the edge congestion. Their running times have a bottlenecking term of $\Omega(m^2\varepsilon^{-2})$, which – as we will see later – arises from a need to run $\widetilde{O}(m\varepsilon^{-2})$ executions of Dijkstra's algorithm.

The main result of this chapter is development of a general approach to speeding up multiplicative-weight-update-based algorithms arising in the context of multicommodity flow. By applying this approach, we will be able to substitute the above $\widetilde{O}(m^2\varepsilon^{-2})$ running time term that all such previous algorithms suffer from, with a $\widetilde{O}(mn\varepsilon^{-2})$ one. This approach is based on two main ideas.

The first one stems from an observation that the shortest-path subproblems that the multiplicative-weight-update-based algorithms solve repeatably are closely related. Namely, each successive subproblem corresponds to the same underlying graph – only the lengths of the arcs differ. Furthermore, the way the lengths of the arcs change is not completely arbitrary – as we will see later, they can only increase over time. This observation suggests that treating each of these subproblems as an independent task – as it is the case in all the previous algorithms – is suboptimal. One might wonder, for example, whether it is possible to maintain a data structure that allows us to answer such a sequence of shortest-path queries more efficiently than just by computing everything from scratch in each iteration. Indeed, it turns out that this kind of questions were already studied extensively in the area of *dynamic graph algorithms* (see e.g. [60, 20, 52, 54, 117, 22, 53, 118, 120, 119, 23]). In particular, the data structure that we would like to maintain in our context corresponds to the *decremental dynamic all-pairs shortest path (DDAPSP)* problem. Unfortunately, this realization alone is not sufficient to obtain the desired improvement, as it turns out that if we are interested in solutions for the DDAPSP problem whose overall running

[2] A *Dantzig-Wolfe-type* algorithm for a fractional packing problem – in which the goal is to find x in some polytope P that satisfies the set of packing inequalities $Ax \leq b$ – is an algorithm that accesses P only by queries of the form: "given a vector c, what is the $x \in P$ minimizing $c \cdot x$?".

Problem	Previous best	Our result
maximum multicommodity flow	$\widetilde{O}(m^2\varepsilon^{-2})$ [65]	$\widetilde{O}(mn\varepsilon^{-2})$
maximum concurrent flow	$\widetilde{O}((m^2+kn)\varepsilon^{-2})$ [84]	$\widetilde{O}((m+k)n\varepsilon^{-2}\log M)$
	$\widetilde{O}(k^2m^2\varepsilon^{-1})$ [109]	

Figure 4-1: Comparison of $(1-\varepsilon)$-approximation schemes for multicommodity flow problems. Here, m is the number of arcs, n is the number of vertices, k is the number of commodities, and $\log M$ is the upper bound on the size of binary representation of any number used in the input instance.

time would be within our intended bounds, then it seems there is no suitable existing result that can be used (see section 4.3 for details).

This lack of existing solution fitting our needs brings us to the second idea. We note that when we employ the multiplicative-weights-update-based framework to solve multicommodity flow problems, it is not necessary to compute the (approximately) shortest path for each shortest-path subproblem. All we really need is that the set of the suitable paths over which we are optimizing the length comes from a set that contains *all* the flowpaths of some *fixed* optimal solution to the multicommodity flow instance that we are solving. To exploit this fact, we introduce a random set of paths $\widehat{\mathcal{P}}$ (formally defined in Definition 4.3.1) that can be seen as a sparsification of the set of all paths in G, and that with high probability has the above-mentioned containment property. Next, we combine the ideas from dynamic graph algorithms to design an efficient data structure that maintains all-pairs shortest path distances with respect to the set $\widehat{\mathcal{P}}$. This data structure allows us to modify (in an almost generic manner) the existing multiplicative-weights-update-based algorithms for various multicommodity flow problems and transform them into Monte-Carlo algorithms with improved running times.

The summary of our results and their comparison with the previous ones can be found in Figure 4-1. To put these result in a context, we note that there are simple examples (see Figure 4-2) that show that no $(1-\varepsilon)$-approximate algorithm for the maximum multicommodity flow problem (resp. maximum concurrent flow problem) can have running time smaller than $O(n^2)$ (resp. $O(n^2+kn)$). On the other hand, in the setting where ε is fixed or moderately small, say $1/\log^{O(1)} n$, and the size of every number used in the input instance is polynomially bounded, our algorithms for these two problems have their running time become $\widetilde{O}(mn)$ and $\widetilde{O}(mn+kn)$ respectively. Thus they are within a $\widetilde{O}(m/n)$ factor of these trivial lower bounds and the $\widetilde{O}(kn)$ term in the second running time is essentially optimal.

Furthermore, these $\widetilde{O}(mn)$ terms almost match a natural barrier for the existing multiplicative-weights-based approaches: the $\Omega(mn)$ *flow-decomposition barrier* corresponding to a *single-commodity* flows. Recall that the flow-decomposition barrier corresponds to the fact that there are instances to the maximum s-t flow problem for whom decomposition of any (even approximately) optimal solution into flowpaths has to have a size of $\Omega(mn)$. As a result, any algorithm that produces such a

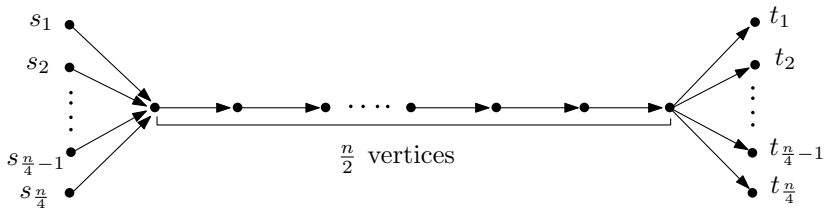


Figure 4-2: In this example, the arcs leaving vertices $\{s_i\}_i$ and entering vertices $\{t_i\}_i$ have capacity 1. The edges on the middle path have capacity $n/4$. By choosing the set of source-sink pairs to be $\{(s_i, t_i)\}_i$ we obtain an instance of the maximum multicommodity flow problem for which any $(1-\varepsilon)$-approximate solution has representation size $\Omega((1-\varepsilon)n^2) = \Omega(n^2)$, for $\varepsilon < \frac{1}{2}$ – each of $n/2-1$ arcs on the middle path has to have non-zero flow of at least $(1-\varepsilon)n/4$ commodities flowing through it. Similarly, by choosing any set of $k \leq n^2/16$ distinct (s_i, t_j) pairs as source-sink pairs for k commodities and setting all demands to 1, we obtain an instance of the maximum concurrent flow problem for which any $(1-\varepsilon)$-approximate solution has representation size $\Omega(kn) = \Omega(n^2 + kn)$ whenever $k \geq n$.

decomposition has to have a running time of $\Omega(mn)$. Therefore, as all the existing multiplicative-weights-based algorithms for the problem are based on iterative finding of shortest paths that are used to flow the commodities, this natural barrier transfers over.

Recall that in the case of the maximum s-t flow problem this barrier was overcome – Goldberg and Rao [73] were the first one to presented an algorithm that improves upon this barrier. Thus it raises an interesting question of whether one can also achieve a similar improvement for the maximum multicommodity flow problem.

4.1.3 Notations and Definitions

In this chapter, we will be dealing with a directed and capacitated graph $G = (V, E, \boldsymbol{u})$, as is our convention, we assume that all capacities are at least one and that U denotes the largest capacity $\max_e u_e$. In addition to capacities, we will often equip arcs of G with lengths described by a vector $\boldsymbol{l}$. For any directed path p in G, by the *length of p with respect to $\boldsymbol{l}$* we mean a quantity $l(p) := \sum_{e \in p} l_e$. For any two vertices u and v of G, a u-v path is a directed path in G that starts at u and ends at v. We define *distance from u to v (with respect to $\boldsymbol{l}$)* for $u, v \in V$ to be the length (with respect to $\boldsymbol{l}$) of the shortest u-v path. We will omit the reference to the length vector $\boldsymbol{l}$ whenever it is clear from the context which length vector we are using.

For $1 \leq i \leq k$, we denote by $\mathcal{P}_i$ the set of all s_i-t_i paths in G, where $\{(s_i, t_i)\}_i$ is the set of k source-sink pairs of the instance of the multicommodity flow problem we are considering. Let $\mathcal{P} = \bigcup_{i=1}^{k} \mathcal{P}_i$. For a given subset $S \subseteq V$ of vertices, let $\mathcal{P}(S)$ be the set of all paths in $\mathcal{P}$ that pass through at least one vertex from S. Finally, for a given $j > 0$, let $\mathcal{P}(S, j)$ be the set of all the paths from $\mathcal{P}(S)$ that consist of at most j arcs.

4.1.4 Outline of This Chapter

We start with section 4.2 where we present the multiplicative-weights-update-based algorithm for the maximum multicommodity flow problem. Next, in section 4.3, we introduce the main ideas and tools behind our results – the connection between fast approximation schemes for multicommodity flow problems and dynamic graph algorithm for maintaining (approximately) shortest paths with respect to the sparsified set of paths. In particular, we formally define the set $\widehat{\mathcal{P}}$, and the $(\delta, M_{\max}, M_{\min}, \widehat{\mathcal{P}})$-ADSP data structure that we will be using for maintenance of the (approximately) shortest paths that we are interested in. Subsequently, in section 4.4, we show how these concepts can be applied to give a more efficient algorithm for the maximum multicommodity flow problem. Then, in section 4.5, we describe an implementation of the $(\delta, M_{\max}, M_{\min}, \widehat{\mathcal{P}})$-ADSP data structure that is efficient enough for our purposes. We conclude in section 4.6 by showing how the ideas developed in this chapter lead to an improved algorithm for the maximum concurrent flow problem.

4.2 Multiplicative-Weights-Update Routine for Solving Maximum Multicommodity Flow Problem

As we mentioned previously, Garg and Könemann [70] presented the first general multiplicative-weights-based framework to solving the multicommodity flow problems, and all the techniques to be presented in this chapter, can be applied in their setting. However, for the sake of illustration, instead of using the framework of [70], we utilize instead the multiplicative-weights-update routine from Section 2.8. As an additional benefit, doing this will also lead to a simpler exposition.

To show how the framework from Section 2.8 can be applied let us consider, for a given value of F, a $(\mathcal{F}, G)$-system in which $\mathcal{F}$ is a set of all s_i-t_i flows of value F in G for each $1 \leq i \leq k$. Note that if one obtains a feasible solution $\{\lambda_f\}_f$ to a $(1+\varepsilon)$-relaxed version of this system then the flow $\overline{f}$ associated with the convex combination $\{\lambda_f\}_f$ can be decomposed into a union of s_i-t_i flows whose value sum up to F and that together exert a congestion of at most $(1+\varepsilon)$ in G. So, this implies that $\overline{f}$ is a multicommodity flow certifying that $F^* \geq F/(1+\varepsilon)$, where $F^* = \sum_i |f_i^*|$ is the value of some optimal solution $f^* = (f_1^*, \ldots, f_k^*)$ to the maximum multicommodity flow problem. On the other hand, if $F \leq F^*$ then f^* corresponds to a feasible solution to the $(\mathcal{F}, G)$-system if one just scales each flow f_i^* down by a factor of F^*/F and includes it in the solution with coefficient $\lambda_{f_i^*} = \frac{|f_i^*|}{\sum_{i'} |f_{i'}^*|}$.

In the light of the above, similarly to the case of maximum s-t flow problem in Chapter 3, we can proceed to relating the task of approximation of the maximum multicommodity flow problem, to the task of designing a $(1 + O(\varepsilon))$-approximate solver for the $(\mathcal{F}, G)$-system described above for a given value of F. To see why this is the case, we just need to note that we can obtain a crude bound on the value of F^* relatively fast. Namely, it is easy to see that

$$B \leq F^* \leq kmB,$$

where B is the maximum bottleneck of any s_i-t_i path in G. (Recall that a bottleneck of a path is the minimum capacity of an edge on that path.) Such a bottleneck B can be computed in $O(mn + n^2 \log n) = \widetilde{O}(mn)$ time, e.g., using the algorithm of Fredman and Tarjan [68]. Now, we can just follow the reasoning outlined in Section 3.3 for the maximum s-t flow problem to reach the desired conclusion.

Therefore, from now on, we can just focus on a particular value of F and the task of designing a $(1+O(\varepsilon))$-approximate solver for the corresponding $(\mathcal{F}, G)$-system. Once again, similarly to the case of Chapter 3, this solver will be based on multiplicative-weights-update routine from Figure 2-1 and thus all we need to do is to design an oracle for it. Our oracle will be very simple. Given a query corresponding to some weights $\boldsymbol{w}$, it considers the length vector $\boldsymbol{l}$ with lengths corresponding to the weights normalized by the capacities, i.e., sets

$$l_e := \frac{w_e}{u_e} \tag{4.1}$$

for each arc e. Then, it finds the shortest s_i-t_i path p with respect to $\boldsymbol{l}$, i.e., the path $p = \operatorname{argmin}_{p \in \mathcal{P}} l(p)$. Once such a path is found, the oracle returns as an answer a flow $\tilde{f}$ that sends F units of flow along the path p, if $l(p) \leq \sum_e l_e u_e / F$; and returns **"fail"**, otherwise.

We claim that this procedure provides a 1-oracle for our $(\mathcal{F}, G)$-system (cf. Definition 2.8.2). To see that this is indeed the case it suffices to prove the following lemma and note that, by definition, $Fl(p) = \sum_e w_e \mathsf{cong}_{\tilde{f}}(e)$ and $\sum_e l_e u_e = \sum_e w_e$.

Lemma 4.2.1. *For any length vector l, whenever $F \leq F^*$, there exists a path $p \in \mathcal{P}$ with $Fl(p) \leq \sum_e l_e u_e$.*

Proof. Let $p_1, \ldots, p_q$ be a decomposition of the optimal solution f^* into flow-paths. The fact that f^* has to obey the capacity constraints implies that

$$\sum_e l_e u_e \geq \sum_{j=1}^{q} l(p_j) f^*(p_j),$$

where $f^*(p_j)$ is the amount of flow flown over the path p_j by f^*. But

$$F \leq F^* = \sum_i |f_i^*| = \sum_{j=1}^{q} f^*(p_j).$$

So, an averaging argument shows that there exists a j^* such that

$$l(p_{j^*}) \leq \sum_e l_e u_e / F.$$

Thus, taking $p = p_{j^*}$ concludes the proof of the Lemma. □

Observe that this oracle can be implemented to run in $\widetilde{O}(km)$ time, by just running Dijkstra's algorithm for each source-sink pair. Therefore, by Theorem 2.8.3 and by

applying Lemma 2.8.9 with the trivial tightness bound of 1, we can conclude that the above multiplicative-weights-update routine makes at most $\widetilde{O}(m\varepsilon^{-2})$ queries to the oracle and thus the following theorem is established.

Theorem 4.2.2 (see also [70]). *For any $1/2 > \varepsilon > 0$, one can compute a $(1-\varepsilon)$-approximation to the maximum multicommodity flow problem in time $\widetilde{O}(km^2\varepsilon^{-2})$.*

Although the above algorithm is conceptually simple, its running time is fairly large – note that for k can be $\Omega(n^2)$ which would result in prohibitively large $\Omega(m^2n^2)$ running time. However, it is not obvious how to reduce this running time significantly[3]. After all, there are limits to how fast the underlying multiple source-sink pairs shortest path problem can be solved (e.g., there is a trivial lower bound of $\Omega(n^2)$) and it is not hard to construct examples on which the bound on the number of iterations is essentially tight.

It turns out, however, that this multiplicative-weights-update-based approach can still be sped up considerably. The key to achieving this is not focusing on finding more efficient way of answering each individual oracle query, but trying to come up instead with a more efficient way of solving the whole sequence of these queries. This insight is pivotal for all the techniques presented in this chapter.

To see an example of how this insight can be utilized, we present a way of improving the running time of the above algorithm that was originally devised by Fleischer [65]. This improvement is based on the realization that whenever we need the shortest path in the above algorithm, it is sufficient to compute a $(1+\varepsilon)$-approximately shortest source-sink path instead of the shortest one. The only thing that is changed by doing so, is that now we are working with a $(1+\varepsilon)$-oracle to the $(\mathcal{F}, G)$-system instead of the 1-oracle that was used above. However, as we are solving this $(\mathcal{F}, G)$-system $(1+\varepsilon)$-approximately anyway, this difference does not change anything.

Now, the fact that providing only $(1+\varepsilon)$-approximate answers to the oracle queries suffices, allows us to modify the way these answers are computed, to avoid solving the shortest-path problem for all the source-sink pairs each time the oracle is queried. The way we achieve this is by cycling through the source-sink pairs, keeping supplying as the answer the shortest path corresponding to the currently chosen source-sink pair as long as the length of this path is at most $(1+\varepsilon)\widehat{\alpha}$ – where $\widehat{\alpha}$ is a maintained by the algorithm lower bound estimate of the current length of the overall shortest path in $\mathcal{P}$ – and moving on to next source-sink pair once it does not. To start, we set $\widehat{\alpha}$ sufficiently small and we do not increase its value as long as we manage to find a path in $\mathcal{P}$ of small enough length (i.e. at most $(1+\varepsilon)\widehat{\alpha}$). Once we are unable to find such a path i.e. our cycling through commodities made a full cycle, we set $\widehat{\alpha} \leftarrow (1+\varepsilon)\widehat{\alpha}$, and start cycling again. The resulting algorithm is presented in Figure 4-3. An important implementation detail used there is that when cycling through source-sink pairs, we group together the pairs that share the same source. This allows us to take advantage of the fact that one execution of Dijkstra's algorithm computes simultaneously the shortest paths for all these pairs.

[3]One possible speeding up would come from noting that one can solve all-pairs shortest path problem in time $O(\min\{mn, n^{2.575}\})$ [66, 137, 140], which gives a speed up in cases of k being very large. But, even after this improvement, the running time is still $\Omega(mn^2)$.

To see why the above modifications reduce the number of shortest path computations needed to execute our multiplicative-weights-update routine, note that each execution of Dijkstra's algorithm either results in providing an answer to oracle query, or causes us to move on in our cycling to the next group of source-sink pairs. Observe that by Lemma 2.8.4 we know that the length of the shortest path is always $n^{O(1/\varepsilon)}$ and thus our way of updating the value of $\widehat{\alpha}$ ensures that there is at most $\lfloor \log_{(1+\varepsilon)} n^{O(1/\varepsilon)} \rfloor = O(\log m\varepsilon^{-2})$ full cycles through the pairs. Also, the number of oracle queries is still $\widetilde{O}(m\varepsilon^{-2})$. Therefore, we have at most $\widetilde{O}((m + \min\{k, n\})\varepsilon^{-2}) = \widetilde{O}(m\varepsilon^{-2})$ executions of Dijkstra's algorithm which leads to the following theorem.

Theorem 4.2.3 (see also [65]). *For any $1/2 > \varepsilon > 0$, the algorithm in Figure 4-3 provides a $(1 - O(\varepsilon))$-approximation to the maximum multi-commodity flow problem in time $\widetilde{O}(m^2\varepsilon^{-2})$.*

4.3 Solving Multicommodity Flow Problems and Dynamic Graph Algorithms

The multiplicative-weights-update-based algorithm for maximum multicommodity flow problem that we described in the previous section, finds a $(1-O(\varepsilon))$-approximate solution by repeatedly solving the subproblem of computing the shortest path in graph G with respect to some length vector $\boldsymbol{l}$ that evolves according to multiplicative weights updates. (Later we will see that the same general approach is also used for another variant of multicommodity flow problem.) As a consequence, the running time of the resulting algorithm is dominated by the time needed to solve these subproblems using Dijkstra's algorithm. As we have seen, by careful choice of the subproblems as well as better utilization of the computed answers, the number of these (single-source) shortest path computations was reduced considerably. However, this number is still $\Omega(m\varepsilon^{-2})$ which leads to a time complexity of $\Omega(m^2\varepsilon^{-2})$ for the corresponding algorithms.

The main observation that underlies the speeding-up techniques we want to develop in this chapter is that treating each of these shortest-path subproblems as an independent task (and thus using Dijkstra's algorithm each time), as was the case so far, is suboptimal. After all, each successive subproblem corresponds to the same underlying graph, only the lengths of some of the arcs increased. Therefore, one might hope to construct a data structure that solves such a sequence of subproblems more efficiently than by computing each time everything from scratch. More precisely, one might wonder whether there is more efficient data structure that maintains a directed graph G with lengths on arcs and supports operations of: increasing a length of some arc; answering shortest-path distance query; and returning shortest vertex-to-vertex path.

It turns out that the problem of designing such a data structure is already known in the literature as the *decremental dynamic all-pairs shortest path* problem and extensive work has been done on this and related problems (see e.g. [60, 20, 52, 54,

Input : Graph $G = (V, E, \boldsymbol{u})$, source-sink pairs $\{(s_i, t_i)\}_i$, accuracy parameter $1/2 > \varepsilon > 0$, and a target flow value F

Output: Either a flow $\overline{f}$, or **"fail"** indicating that $F > F^*$

Initialize the multiplicative-weights-update routine D from Figure 2-1 (for $\mathcal{F}$ being the set of s_i-t_i flows of value F in G), let $\boldsymbol{w}$ be the corresponding weights vector
Initialize $l_e \leftarrow w_e/u_e$ for each arc $e \in E$ (cf. (4.1)), $\widehat{\alpha} \leftarrow 1/U$, and
$I(s) \leftarrow \{i \mid s_i = s\}$ for each $s \in V$ (* grouping source-sink pairs with common sources *)
while True **do**
  foreach $s \in V$ with $I(s) \neq \emptyset$ **do** (* cycling through commodities *)
    Use Dijkstra's algorithm to find the shortest path p in $\bigcup_{i \in I(s)} \mathcal{P}_i$
    while $l(p) < (1+\varepsilon)\widehat{\alpha}$ **do**
      if $l(p) > (1+\varepsilon) \sum_e l_e u_e / F$ **then return** ***"fail"***
      else
        Supply to D the flow $\tilde{f}$ sending F units of flow over p as an oracle answer
        Update the weights $\boldsymbol{w}$ as dictated by the routine D
        Update the lengths $\boldsymbol{l}$ to make (4.1) hold with respect to updated weights
        if D terminates **then return** the flow $\overline{f}$ computed by D
      end
    end
    Use Dijkstra's algorithm to find the shortest path p in $\bigcup_{i \in I(s)} \mathcal{P}_i$
  end
end
$\widehat{\alpha} \leftarrow (1+\varepsilon)\widehat{\alpha}$ (* increase the value of $\widehat{\alpha}$ *)

Figure 4-3: Faster algorithm for computing $(1 - O(\varepsilon))$-approximation to the maximum multicommodity flow problem

117, 22, 53, 118, 120, 119, 23]). However, if we are interested in solutions whose overall running time is within our intended bounds, the result that is closest to what we need is the one by Roditty and Zwick [118]. They show that if G were *undirected* and had positive, *integer* lengths on edges, with maximum length being b, a $(1+\delta)$-approximate solution to the decremental dynamic all-pairs shortest path problem can be implemented with total maintenance time of $\widetilde{O}(\frac{mnb}{\delta})$, $O(1)$ time needed to answer any vertex-to-vertex shortest-path distance query, and returning shortest vertex-to-vertex path in $O(n)$ time.[4] Unfortunately, in our applications the graph G is not only directed, but also the lengths of the arcs can be of order of $b = \Omega(n^{1/\varepsilon})$ with $\varepsilon < 1/2$ (cf. Lemma 2.8.4. Therefore, the resulting running time would be prohibitive. Further, the construction of Roditty and Zwick assumes that the sequence of operations to be handled is oblivious to the behavior of the data structure (e.g. to its randomized choices). This feature is another shortcoming from our point of view since in our setting the way the lengths of the arcs evolves depends directly on which shortest paths the data structure chose to output previously.

To circumvent this lack of suitable existing solutions, we observe that for our purposes it is sufficient to solve a simpler task than the decremental dynamic all-pairs shortest path problem in its full generality (i.e. in the directed setting, allowing large lengths on arcs and adversarial requests). Namely, when apply the multiplicative-weights-update-based approach from the previous section, it is not necessary to compute for each of the resulting subproblems the (approximately) shortest path among all the suitable paths in the set $\mathcal{P}$. As we will prove later, to establish satisfactory bounds on the quality of the final flow, it suffices that whenever we solve some shortest-path subproblem, the set of suitable paths over which we are optimizing the length comes from a set that contains *all* the flowpaths of some fixed optimal solution to the instance of the multicommodity flow problem that we are solving. To see why this is the case, we just need to note that Lemma 4.2.1 – that we need to establish to obtain the desired bound on the quality of the return solution – is proved by only considering flowpaths of some optimal solution.

With this observation in mind, we define the following random subset $\widehat{\mathcal{P}}$ of paths in $\mathcal{P}$.[5] One may view $\widehat{\mathcal{P}}$ as a sparsification of the set $\mathcal{P}$.

Definition 4.3.1. *For $j = 1, \ldots, \lceil \log n \rceil$, let S_j be a random set obtained by sampling each vertex of V with probability $p_j = \min\{\frac{10 \ln n}{2^j}, 1\}$. Define $\widehat{\mathcal{P}} := \bigcup_{j=1}^{\lceil \log n \rceil} \mathcal{P}(S_j, 2^j)$, where $\mathcal{P}(S, j')$ for a given $S \subseteq V$, and $j' > 0$ is the set of all paths in $\mathcal{P}$ that consist of at most j' arcs and pass through at least one vertex of S.*

It turns out that this sparsification $\widehat{\mathcal{P}}$ retains, with high probability, the key property that we need.

[4]Note that there can be as many as mb edge length increases in a sequence, thus this solution is faster than naïve solution that just computes all-pairs shortest-path after each edge-length increase, or one that just answers each of the queries using Dijkstra's algorithm.

[5]It is worth noting that a very similar set was used in [118] albeit with a slightly different motivation.

Lemma 4.3.2. *For any fixed multicommodity flow solution $f = (f_1, \ldots, f_k)$, with high probability, all the flowpaths of f are contained in $\widehat{\mathcal{P}}$.*

Proof. Let $p_1, \ldots p_q$ be the decomposition of the flow f into flowpaths. By definition, all p_i are contained in the set $\mathcal{P}$. Furthermore, by standard flow decomposition argument we know that $q \leq km \leq n^4$. Let us focus now on some particular path p_i. Let $1 \leq t \leq n$ be the number of arcs in this path, and let j^* be the smallest j for which $t \leq 2^j$. The probability that $p_i \in \mathcal{P}(S_{j^*}, 2^{j^*}) \subseteq \widehat{\mathcal{P}}$ is exactly the probability that at least one vertex from p is in S_{j^*}. Simple computation shows that the probability that none among $t+1$ vertices of p_i is in $\mathcal{P}(S_{j^*}, 2^{j^*})$ is at most

$$(1 - \frac{10 \ln n}{2^{j^*}})^{t+1} \leq e^{-\frac{10(t+1)\ln n}{2^{j^*}}} \leq e^{-5 \ln n} = n^{-5}.$$

Therefore, by union bounding over all $q \leq n^4$ paths p_i, we get that indeed $\{p_1, \ldots, p_q\} \subseteq \widehat{\mathcal{P}}$ with high probability. □

4.3.1 $(\delta, M_{\max}, M_{\min}, \mathcal{Q})$-ADSP data structure

Once we defined set $\widehat{\mathcal{P}}$, our goal is to devise an efficient way of maintaining the $(1+\delta)$-approximate shortest paths with respect to it. We start by formally defining our task.

Definition 4.3.3. *For any $\delta \geq 0$, $M_{\max} \geq 2M_{\min} > 0$ and a set of paths $\mathcal{Q} \subseteq \mathcal{P}$, let the* δ-approximate decremental $(M_{\max}, M_{\min}, \mathcal{Q})$-shortest path problem *($(\delta, M_{\max}, M_{\min}, \mathcal{Q})$-ADSP, for short) be a problem in which one maintains a directed graph G with length vector $\boldsymbol{l}$ that supports four operations (sometimes we will refer to these operations as* requests*):*

- *Distance(u, v, β), for $u, v \in V$, and $\beta \in [M_{\min}, M_{\max}/2]$: let d^* be the length of the shortest (with respect to l) u-v path in $\mathcal{Q}$; if $d^* \leq 2\beta$ then the query returns a value d such that $d \leq d^* + \delta\beta$. If $d^* > 2\beta$, the query may return either d as above, or ∞;*
- *Increase(e, ω), for $e \in E$ and $\omega \geq 0$: increases the length $l(e)$ of the arc e by ω;*
- *Path(u, v, β), for $u, v \in V$, and $\beta \in [M_{\min}, M_{\max}/2]$: returns a u-v path of length at most Distance(u, v, β), as long as Distance$(u, v, \beta) \neq \infty$;*
- *SSrcDist(u, β) for $u \in V$, and $\beta \in [M_{\min}, M_{\max}/2]$: returns Distance$(u, v, \beta)$ for all $v \in V$.*

Intuitively, β is our guess on the interval $[\beta, 2\beta]$ in which the length of the shortest path we are interested in is. We say that β is (u, v)-*accurate* for given $(\delta, M_{\max}, M_{\min}, \mathcal{Q})$-ADSP data structure R and $u, v \in V$, if the length d^* of the shortest u-v path in $\mathcal{Q}$ is at least β and the data structure R returns a finite value in response to Distance(u, v, β) query. Note that if β is (u, v)-accurate then the $\delta\beta$-additive error guarantee on the distance estimation supplied by R in response to

Distance(u, v, β) query implies a $(1+\delta)$ *multiplicative* error guarantee. Also, as long as d^* is at least $M_{\min}$ (this will be always the case in our applications), we can employ binary search to ask $O(\log\log M_{\max}/M_{\min})$ Distance$(u, v, \cdot)$ requests and either realize that d^* is bigger than $M_{\max}$, or find $0 \leq i \leq \lceil \log M_{\max}/2M_{\min} \rceil$ such that $\beta_i = \min\{2^i M_{\min}, M_{\max}/2\}$ is (u, v)-accurate.[6] Finally, it is worth emphasizing that we do not require that the paths returned in response to Path$(\cdot,\cdot,\cdot)$ requests are from $\mathcal{Q}$ – all we insist on is just that the length of all the suitable paths from $\mathcal{Q}$ are considered when the path to be returned is chosen.

In section 4.5, we will describe how the ideas and tools from dynamic graph algorithms lead to an implementation of the $(\delta, M_{\max}, M_{\min}, \widehat{\mathcal{P}})$-ADSP data structure that is tailored to maintain the shortest paths from set $\widehat{\mathcal{P}}$ and whose performance is described in the following theorem. The theorem is proved in section 4.5.2.

Theorem 4.3.4. *For any $1 > \delta > 0$, $M_{\max} \geq 2M_{\min} > 0$, $(\delta, M_{\max}, M_{\min}, \widehat{\mathcal{P}})$-ADSP data structure can be maintained in total expected time $\widetilde{O}(mn\frac{\log M_{\max}/M_{\min}}{\delta})$ plus additional $O(1)$ per* Increase$(\cdot,\cdot)$ *request in the processed sequence. Each* Distance$(\cdot,\cdot,\cdot)$ *and* Path$(\cdot,\cdot,\cdot)$ *query can be answered in $\widetilde{O}(n)$ time, and each* SSrcDist$(\cdot,\cdot)$ *query in $\widetilde{O}(m)$ time.*

4.3.2 Solving the Decremental Dynamic All-Pairs Shortest Paths Problem Using the $(\delta, M_{\max}, M_{\min}, \widehat{\mathcal{P}})$-ADSP Data Structure

Before we proceed further, we want to note that even though the $(\delta, M_{\max}, M_{\min}, \widehat{\mathcal{P}})$-ADSP data structure construction from Theorem 4.3.4 was designed with the aim of speeding up the multicommodity flow algorithms, it turns out it also establishes a new result for the dynamic shortest-path algorithms. Namely, using Theorem 4.3.4 one can obtain a $(1+\varepsilon)$-approximate solution to the oblivious decremental dynamic all-pairs shortest path problem in directed graphs with rational arc lengths, where obliviousness means that the sequence of requests that we process does not depend on the randomness used in the solution.

Theorem 4.3.5. *For any $1 > \varepsilon > 0$, and $L \geq 1$ there exists a $(1+\varepsilon)$-approximate Monte Carlo solution to the oblivious decremental dynamic all-pairs shortest paths problem on directed graphs where arc lengths are rational numbers between 1 and L, that has total maintenance cost of $\widetilde{O}(mn\frac{\log L}{\varepsilon})$ plus additional $O(1)$ per increase of the length of any arc, and answers shortest path queries for any vertex pair in $\widetilde{O}(n(\log\log_{(1+\varepsilon)} L)(\log\log L))$ time.*

Note that even when we allow lengths to be quite large (e.g. polynomial in n), the maintenance cost of our solution is still similar to the one that Roditty and Zwick

[6] The binary search just chooses i in the middle of the current range of values in which the desired value of i may lie (initially this range is $[0, \lceil \log M_{\max}/2M_{\min} \rceil]$), and if the Distance$(u, v, \beta_i)$ query returns ∞ then the left half of the range (including the i queried) is dropped, otherwise the other half is dropped.

achieved in [118] for undirected graphs with small integer lengths. Unfortunately, our distance query time is $\widetilde{O}(n)$ instead of the $O(1)$ time obtained in [118]. So, the gain that we get over a naïve solution for the problem is that we are able to answer ($(1+\varepsilon)$-approximately) shortest path queries for any vertex pair in $\widetilde{O}(n)$ time, as opposed to the $O(m + n \log n)$ time required by Dijkstra's algorithm.

Proof of Theorem 4.3.5. Let $G = (V, E)$ be the graph of our interest, and let $\boldsymbol{l}^i$ be the length vector of G after processing i requests from our sequence. Consider a u-v path p, for some $u, v \in V$, that is the shortest u-v path in G with respect to $\boldsymbol{l}^i$, for some i. Our construction is based on the following simple observation: for any $\delta > 0$, and for all $i' \geq i$, as long as for all arcs e of p, $l_e^{i'}$ is at most $(1+\delta)l_e^i$, p remains to be a $(1+\delta)$-approximate shortest u-v path in G with respect to $\boldsymbol{l}^{i'}$. Now, note that we have n^2 different (u, v) pairs, and each of m arcs can increase its length by a factor $(1+\delta)$ at most $\lceil \log_{(1+\delta)} L \rceil$ times. Therefore this observation implies that for any $\delta > 0$ there exists a set $\mathcal{Q}(\delta)$ of $O(mn^2 \frac{\log L}{\delta})$ paths such that for any (u, v) and i there exists a u-v path p in $\mathcal{Q}(\delta)$ that has length $l^i(p)$ within $(1+\delta)$ of the length $l^i(p^*)$ of the shortest (with respect to $\boldsymbol{l}^i$) u-v path p^* in G.

In the light of the above, our solution for the decremental all-pairs shortest path problem is based on maintaining $(\varepsilon/3, Ln, 1, \widehat{\mathcal{Q}}(\varepsilon/3))$-ADSP data structure R, where $\widehat{\mathcal{Q}}(\varepsilon/3)$ is the set constructed as in Definition 4.3.1 after we make $\mathcal{P}$ to be the set of all paths in G and change the sampling probabilities p_j to $\min\{\frac{10 \ln(n \lceil \log_{(1+\varepsilon/3)} L \rceil)}{2^j}, 1\}$ for $j = 1, \ldots, \lceil \log n \rceil$. Note that by reasoning completely analogous to the one from the proof of Lemma 4.3.2, we can argue that with high probability $\mathcal{Q}(\varepsilon/3) \subseteq \widehat{\mathcal{Q}}(\varepsilon/3)$. Also, by straight-forward adjustment of the construction from Theorem 4.3.4 we obtain an implementation of $(\varepsilon/3, Ln, 1, \widehat{\mathcal{Q}}(\varepsilon/3))$-ADSP data structure that has total maintenance cost of $\widetilde{O}(mn \frac{\log L}{\varepsilon})$ plus $O(1)$ time per each $\mathsf{Increase}(\cdot, \cdot)$ request, and that answers $\mathsf{Path}(\cdot, \cdot, \cdot)$ and $\mathsf{Distance}(\cdot, \cdot, \cdot)$ requests in $\widetilde{O}(n \log \log_{(1+\varepsilon)} L)$ time. Therefore, if we process the request sequence by just passing arc length increase requests to R, and answering each u-v shortest path query by issuing to R a $\mathsf{Path}(u, v, \beta)$ query – where (u, v)-accurate β is found through binary search using $O(\log \log L)$ $\mathsf{Distance}(u, v, \cdot)$ requests (see discussion after Definition 4.3.3), then the Definition 4.3.3 ensures that we obtain a correct $(1+\varepsilon)$-approximate solution for decremental all-pairs shortest path problem whose performance obeys the desired bounds. The theorem follows. □

4.4 Maximum Multicommodity Flow Problem

We proceed to developing a faster algorithm for the maximum multicommodity flow problem. As we indicated in the previous section, the basic idea behind our improvement is making the procedure presented in Section 4.2 (see Figure 4-3) to exploit the dependencies between the shortest-path subproblems it is solving. More precisely, instead of employing Dijkstra's algorithm each time, we answer these shortest-path questions by querying a $(\delta, M_{\max}, M_{\min}, \widehat{\mathcal{P}})$-ADSP data structure (as described in Theorem 4.3.4) that we maintain for an appropriate choice of δ, $M_{\max}$, and $M_{\min}$,

and where $\widehat{\mathcal{P}}$ is a sparsification of $\mathcal{P}$, as described in Definition 4.3.1. However, the straight-forward implementation of this idea encounters some difficulties.

First, we have to justify the fact that while answering shortest-path queries we take into account mainly paths in $\widehat{\mathcal{P}}$, as opposed to the whole set $\mathcal{P}$. Second, an important feature of Dijkstra's algorithm that is exploited in Fleischer's approximation scheme, is the fact that whenever one computes the distance between a pair of vertices using this algorithm it simultaneously computes all single source shortest-path distances. Unfortunately, in our case we cannot afford to replicate this approach – the SSrcDist$(\cdot,\cdot)$ query that would be an equivalent of Dijkstra's algorithm's execution here, takes still $\widetilde{O}(m)$ instead of desired $\widetilde{O}(n)$ time; we need to circumvent this issue in a more careful manner. We address both these problems below.

4.4.1 Existence of Short Paths in $\widehat{\mathcal{P}}$

As mentioned in section 4.2, the key ingredient used in our multiplicative-weights-update-based framework to bound the quality of the solution for the maximum multicommodity flow problem it obtains, is Lemma 4.2.1. We prove now that this Lemma still holds, with high probability, that the same property still holds when we consider only paths in $\widehat{\mathcal{P}}$.

Lemma 4.4.1. *With high probability, for any length vector l, whenever $F \leq F^*$, there exists a path $p \in \widehat{\mathcal{P}}$ with $Fl(p) \leq \sum_e l_e u_e$.*

Proof. Let $f^* = (f_1^*, f_2^*, \ldots, f_k^*)$ be some optimal multicommodity flow with $\sum_i |f_i^*| = F^*$. By Lemma 4.3.2 we know that with high probability $\widehat{\mathcal{P}}$ contains all the flow-paths $p_1, \ldots, p_q$ of f^*. The fact that f^* has to obey the capacity constraints implies that $\sum_e l_e u_e \geq \sum_{j=1}^{q} l(p_j) f^*(p_j)$. But $OPT = \sum_i |f_i^*| = \sum_{j=1}^{q} f^*(p_j)$; an averaging argument shows that there exists a j^* such that $l(p_{j^*}) \leq \sum_e l_e u_e / F$ as desired. □

4.4.2 Randomized Cycling Through Commodities

For a given value of $\widehat{\alpha}$, and some $(\delta, M_{\max}, M_{\min}, \widehat{\mathcal{P}})$-ADSP data structure R, we say that a source-sink pair (s,t) is *admissible for $\widehat{\alpha}$ (with respect to R)* if upon querying R with Distance$(s,t,\widehat{\alpha})$ the obtained answer is at most $(1+2\delta)\widehat{\alpha}$. In other words, (s,t) is admissible for $\widehat{\alpha}$ if R's estimate of the distance from s to t in $\widehat{\mathcal{P}}$ is small enough that our algorithm could choose to augment the flow along this path (provided $\widehat{\alpha}$ was its current lower-bound estimate of the length of the shortest path in $\widehat{\mathcal{P}}$). Obviously, our algorithm is vitally interested in finding source-sink pairs that are admissible for its current value of $\widehat{\alpha}$ – these pairs are the ones that allow us to answer the oracle queries.

Unfortunately, given the set of all possible pairs $\{(s_1,t_1), \ldots, (s_k,t_k)\}$, and the data structure R, it is not clear at all which one among them (if any) are admissible for given $\widehat{\alpha}$. Note, however, that if we deem some source-sink pair (s,t) inadmissible for $\widehat{\alpha}$ (by querying R for the corresponding s-t distance) then, since our lengths are always increasing, this pair will never become admissible for $\widehat{\alpha}$ again. This suggests

the following natural approach for identification of admissible pairs for a given $\widehat{\alpha}$. We cycle through all the sink-source pairs and query R for the corresponding distance, we stick with one pair as long as it is admissible, and move on once it becomes inadmissible. Clearly, the moment we cycled through all pairs, we know that all the pairs are inadmissible for $\widehat{\alpha}$. The problem with this approach is that the resulting number of s-t distance queries is at least k and thus this would lead to the somewhat prohibitive bottleneck of $\widetilde{\Omega}(kn)$ in the running time (note that k can be $\Omega(n^2)$).

To alleviate this problem we note that a very similar issue was present already in the algorithm from Section 4.2 – the way it was avoided there was by grouping the source-sink pairs according to common sources and exploiting the fact that Dijkstra's algorithm computes all single source shortest-path distances simultaneously. Therefore, one execution of Dijkstra's algorithm either finds an (analog of) admissible pair, or deems *all* the pairs sharing the same source inadmissible. Unfortunately, although our $(\delta, M_{\max}, M_{\min}, \widehat{\mathcal{P}})$-ADSP data structure allows single source shortest-path distance queries, these queries require $\widetilde{O}(m)$ time and we cannot afford to use them to obtain s-t distances in the manner it was done in Section 4.2 – this could cause $\Omega(m^2\varepsilon^{-2})$ worst-case running time. We therefore devise a different method of circumventing this bottleneck. To describe it, let us assume that we are given some vertex s, and a set $I(s)$ of source-sink pairs that have s as their common source and that have not yet been deemed inadmissible for our current value of $\widehat{\alpha}$. Our procedure samples $\lceil \log n \rceil$ source-sink pairs from $I(s)$ and checks whether any of them is admissible using the $\mathsf{Distance}(\cdot,\cdot,\cdot)$ query. If so, then we return the admissible pair found. Otherwise, i.e. if none of them was admissible, we use the $\mathsf{SSrcDist}(\cdot,\cdot)$ query to check which (if any) source-sink pairs in $I(s)$ are inadmissible, remove them from the set $I(s)$ and return an admissible pair (if any was found). We repeat the whole procedure – if $I(s)$ became empty, we proceed to the next possible source s – until we examine all source-sink pairs. The algorithm is summarized in Figure 4-5 – for convenience, we substituted for δ, $M_{\max}$, and $M_{\min}$ the actual values used in our algorithm. The intuition behind this procedure is that if all $\lceil \log n \rceil$ samples from $I(s)$ turned out to be inadmissible then with probability at least $(1 - \frac{1}{n})$, at least half of the pairs in $I(s)$ is inadmissible, and therefore the $\mathsf{SSrcDist}(\cdot,\cdot)$ query will reduce the size of $I(s)$ by at least half. This leads to the expected number of $\mathsf{SSrcDist}(\cdot,\cdot)$ requests being not too large.

4.4.3 Our Algorithm

We present our algorithm for the maximum multicommodity flow problem in Figure 4-4. As already noted, the basic idea behind it is making the algorithm from Section 4.2 (presented in Figure 4-3) to answer shortest-path distance queries using an $(\varepsilon/2, 1/U, n^{2(1+\varepsilon)/\varepsilon}, \widehat{\mathcal{P}})$-ADSP data structure (where $\widehat{\mathcal{P}}$ is constructed as in Definition 4.3.1). To implement this idea efficiently, we incorporated the randomized cycling through commodities described above.

We prove the following theorem.

Input : Graph $G = (V, E, \boldsymbol{u})$, source-sink pairs $\{(s_i, t_i)\}_i$, accuracy parameter $1/2 > \varepsilon > 0$, and a target flow value F

Output: Either a flow $\overline{f}$, or **"fail"** indicating that $F > F^*$

Initialize the multiplicative-weights-update routine D from Figure 2-1 (for $\mathcal{F}$ being the set of s_i-t_i flows of value F in G), let $\boldsymbol{w}$ be the corresponding weights vector
Initialize $l_e \leftarrow w_e/u_e$ for each arc $e \in E$ (cf. (4.1)), $\widehat{\alpha} \leftarrow 1/U$, and
$I(s) \leftarrow \{i \mid s_i = s\}$ for each $s \in V$ (* grouping source-sink pairs with common sources *)
Sample sets $\{S_j\}_{j=1,\ldots,\lceil \log n \rceil}$ to indicate the set $\widehat{\mathcal{P}}$ (see Definition 4.3.1)
Initialize $(\varepsilon/2, 1/U, n^{2(1+\varepsilon)/\varepsilon}, \widehat{\mathcal{P}})$-ADSP data structure R as in Theorem 4.3.4
while True **do**
  $I(s) \leftarrow \{i \mid s_i = s\}$ for each $s \in V$ (* initializing pairs to be examined *)
  foreach $s \in V$ with $I(s) \neq \emptyset$ **do** (* cycling through commodities *)
    $\langle (s,t), I(s) \rangle \leftarrow$ Find_Admissible_Pair$(\varepsilon, R, \widehat{\alpha}, I(s))$
    while $I(v) \neq \emptyset$ **do**
      $p \leftarrow R$.Path$(s, t, \widehat{\alpha})$
      if $l(p) > (1+\varepsilon) \sum_e l_e u_e / F$ **then return** ***"fail"***
      else
        Supply to D the flow $\tilde{f}$ sending F units of flow over p as an oracle answer
        Update the weights $\boldsymbol{w}$ as dictated by the routine D
        Use R.Increase$(\cdot, \cdot)$ requests to make the lengths $\boldsymbol{l}$ obey (4.1) with respect to updated weights
        if R terminates **then return** the flow $\overline{f}$ computed by D
        $\langle (s,t), I(s) \rangle \leftarrow$ Find_Admissible_Pair$(\varepsilon, R, \widehat{\alpha}, I(s))$
      end
    end
    $\widehat{\alpha} \leftarrow (1+\varepsilon/2)\widehat{\alpha}$ (* increase the value of $\widehat{\alpha}$ *)
  end
end

Figure 4-4: Improved algorithm for solving maximum multicommodity flow. Find_Admissible_Pair procedure is described in Figure 4-5.

Input : Accuracy parameter $\varepsilon > 0$, $(\varepsilon/2, 1/U, n^{2(1+\varepsilon)/\varepsilon}, \widehat{\mathcal{P}})$-ADSP data structure R, lower bound $\widehat{\alpha}$ on $\min_{p \in \widehat{\mathcal{P}}} l(p)$, and a set $I(s)$ of source-sink pairs that might be admissible for $\widehat{\alpha}$

Output: $\langle (s,t), I'(s) \rangle$, where $I'(s)$ is a subset of $I(s)$, and (s,t) is: an admissible pair for $\widehat{\alpha}$ if $I'(s) \neq \emptyset$; an arbitrary pair otherwise

for $i = 1, \ldots, \lceil \log n \rceil$ **do**
  Sample a random source-sink pair (s,t) from $I(s)$
  if $R.\mathit{Distance}(s,t,\widehat{\alpha}) \leq (1+\varepsilon)\widehat{\alpha}$ **then** (* `checking admissibility for` $\widehat{\alpha}$ *)
    return $\langle (s,t), I(s) \rangle$
  end
end
(* `No admissible pairs sampled` *)
Use R.SSrcDist$(s, \widehat{\alpha})$ to check admissibility for $\widehat{\alpha}$ of all the source-sink pairs in $I(s)$
Let $I'(s)$ be the subset of admissible pairs from $I(s)$
if $I'(s) \neq \emptyset$ **then**
  return $\langle (s,t), I'(s) \rangle$ where $(s,t) \in I'(s)$
else
  return $\langle (s,t), \emptyset \rangle$ where (s,t) is an arbitrary pair
end

Figure 4-5: Procedure Find_Admissible_Pair

Theorem 4.4.2. *For any $1/2 > \varepsilon > 0$, with high probability, one can obtain a $(1 - O(\varepsilon))$-approximate solution to the maximum multicommodity flow problem in expected $\widetilde{O}(mn\varepsilon^{-2})$ time.*

Proof. By our discussion at the beginning of Section 4.2, we just need to prove that, whenever $F \leq F^*$, the algorithm presented in Figure 4-4 returns a flow $\overline{f}$ that, with high probability, exerts a congestion of at most $(1+\varepsilon)$ in G, and that it works in expected time of $\widetilde{O}(mn\varepsilon^{-2})$.

To this end, let us first notice that as we are explicitly check whether $l(p) \leq (1+\varepsilon)\sum_e l_e u_e / F$, we know that whenever we pass a flow $\tilde{f}$ corresponding to the path p, to the routine D, it is a correct answer of a $(1+\varepsilon)$-oracle. As a result, we can use Lemma 2.8.4 to prove that the weights we are dealing with are never bigger than $n^{2(1+\varepsilon)/\varepsilon}$, and thus our value of $M_{\max}$ is initialized correctly (as is $M_{\min}$ since $l_e \geq 1/U = M_{\min}$ for each e). Furthermore, by Theorem 2.8.3, we know that all we have to do is to prove that our algorithm never returns **"fail"** when $F \leq F^*$ and that it never takes more than $\widetilde{O}(mn\varepsilon^{-2})$ steps to terminate.

In order to establish the first fact, let $\boldsymbol{l}^j$ be the length vector $\boldsymbol{l}$ after j-th oracle answer has been computed, and let α_j be the length of the shortest path in $\widehat{\mathcal{P}}$ with respect to l^j. We want to prove that for every j, the value $\widehat{\alpha}_j$ of $\widehat{\alpha}$ immediately before $(j+1)$-th augmentation is at most α_j. Once we do that, then the definition of $(\varepsilon/2, 1/U, n^{2(1+\varepsilon)/\varepsilon}, \widehat{\mathcal{P}})$-ADSP and Lemma 4.4.1, will allow us to conclude that the algorithm never returns **"fail"** for $F \leq F^*$.

To prove that $\widehat{\alpha}_j \leq \alpha_j$ for each j, assume for the sake of contradiction that this

is not the case; let j^* be the first j for which $\widehat{\alpha}_j > \alpha_j$. By definition, this means that there exists an s-t path $p \in \widehat{\mathcal{P}}$ with $l^{j^*}(p) < \widehat{\alpha}_{j^*}$. Since $l_{j^*}(p) \geq \min_e l_e = 1/U = \widehat{\alpha}_0$, it must have been the case that $\widehat{\alpha}_{j^*} > \gamma$ and the pair (s,t) was deemed inadmissible for $\frac{\widehat{\alpha}_{j^*}}{(1+\varepsilon/2)}$ at some earlier point of the algorithm – otherwise the value of $\widehat{\alpha}$ would have never increased up to $\widehat{\alpha}_{j^*}$. But this is impossible, since the length of p could only increase over time and

$$l^{j^*}(p) + \frac{\varepsilon}{2}\frac{\widehat{\alpha}_{j^*}}{(1+\varepsilon/2)} < \widehat{\alpha}_{j^*} + \frac{\varepsilon}{2}\frac{\widehat{\alpha}_{j^*}}{(1+\varepsilon/2)} = (1+\varepsilon)\frac{\widehat{\alpha}_{j^*}}{(1+\varepsilon/2)}.$$

Thus by Theorem 4.3.4, we know that $\mathsf{Distance}(s, t, \widehat{\alpha}_{j^*}/(1+\varepsilon/2))$ must have had return a value of at most $(1+\varepsilon)\frac{\widehat{\alpha}_{j^*}}{(1+\varepsilon/2)}$ which would deem the pair (s,t) admissible. This contradiction gives us the desired claim and proves the correctness of our algorithm.

Now, to bound the running time of the algorithm, we note that it is dominated by the total cost of maintaining our $(\varepsilon/2, 1/U, n^{2(1+\varepsilon)/\varepsilon}, \widehat{\mathcal{P}})$-ADSP data structure R, and servicing all the requests issued to it – all other operations performed by the algorithm can be amortized within this total cost.

Note that by the reasoning analogous to the one we did in case of the maximum s-t flow problem, we can assume that the largest capacity U (after normalizing the smallest one to be equal to 1) is bounded by a polynomial in m and $1/\varepsilon$. Now, by Theorem 4.3.4, we obtain that the total expected maintenance cost of our data structure R is at most $\widetilde{O}(mn\frac{\log n^{2(1+\varepsilon)/\varepsilon} U}{\varepsilon}) = \widetilde{O}(mn\varepsilon^{-2})$. Also, note that by Lemma 2.8.9 (taking tightness to be 1), there can be at most $\widetilde{O}(m\varepsilon^{-2})$ oracle answers.

As a result, the cost of serving all the $\mathsf{Path}(\cdot,\cdot,\cdot)$ request is at most $\widetilde{O}(mn\varepsilon^{-2})$, as desired. Furthermore, as each of the paths p returned can be assumed to consist of at most n arcs and the only length that increase after the flow corresponding to p is supplied to the routine D are the ones being in p, the total cost of serving $\mathsf{Increase}(\cdot,\cdot)$ request is at most $\widetilde{O}(m\varepsilon^{-2}) \cdot n \cdot O(1) = \widetilde{O}(mn\varepsilon^{-2})$ too. and $\mathsf{Increase}(\cdot,\cdot)$ requests is at most $\widetilde{O}(m\varepsilon^{-2}(\widetilde{O}(n) + n \cdot O(1))) = \widetilde{O}(mn\varepsilon^{-2})$, as desired.

Now, to bound the time needed to service all the $\mathsf{Distance}(\cdot,\cdot,\cdot)$ requests, we note that there can be at most $\widetilde{O}(m\varepsilon^{-2})$ samplings of source-sink pairs during the $\mathsf{Find_Admissible_Pair}$ procedure that yield an admissible pair. This is so, since finding an admissible pair results in an oracle answer. Thus the total cost of samplings that found some admissible pair is at most $\lceil \log n \rceil \widetilde{O}(m\varepsilon^{-2})\widetilde{O}(n) = \widetilde{O}(mn\varepsilon^{-2})$. On the other hand, in cases when sampling does not yield an admissible path the time needed to serve all the corresponding $\mathsf{Distance}(\cdot,\cdot,\cdot)$ requests can be amortized in the time needed to serve $\mathsf{SSrcDist}(\cdot,\cdot)$ that is always issued afterward.

Therefore, all that is left to do is to bound the expected service cost of $\mathsf{SSrcDist}(\cdot,\cdot)$ requests. We call a $\mathsf{SSrcDist}(s,\widehat{\alpha})$ *successful* if it resulted in decreasing the size of $I(s)$ by at least a half. Note that there can be at most $\lceil \log n \rceil$ successful $\mathsf{SSrcDist}(\cdot,\cdot)$ requests per one source and one value of $\widehat{\alpha}$. As a result, the total time spent on answering successful requests is at most $\log_{(1+\varepsilon/2)}(n^{2(1+\varepsilon)/\varepsilon}U) \cdot \widetilde{O}(m) = \widetilde{O}(mn\varepsilon^{-2})$, since we have at most $\log_{(1+\varepsilon/2)}(n^{2(1+\varepsilon)/\varepsilon}U)$ different values of $\widehat{\alpha}$. On the other hand,

to bound the time taken by serving unsuccessful $\mathsf{SSrcDist}(\cdot,\cdot)$ requests, we notice that each (successful or not) $\mathsf{SSrcDist}(\cdot,\cdot)$ query either empties one set $I(s)$ for given source s and a value of $\widehat{\alpha}$, or finds an admissible pair (which results in flow augmentation), therefore there can be at most $\widetilde{O}(m\varepsilon^{-2})$ $\mathsf{SSrcDist}(\cdot,\cdot)$ requests in total. Moreover, the probability that a particular query is unsuccessful is at most $\frac{1}{n}$ - this follows from the fact that if $I(s)$ has more than a half of pairs admissible for given $\widehat{\alpha}$ then the probability that none among $\lceil \log n \rceil$ independent samples turns out to be admissible is at most $(\frac{1}{2})^{\lceil \log n \rceil} \leq \frac{1}{n}$. Therefore, the total expected cost of this type of requests is at most $\frac{\widetilde{O}(mn\varepsilon^{-2}}{n} \cdot \widetilde{O}(m) = \widetilde{O}(\frac{m^2}{n\varepsilon^2}) = \widetilde{O}(mn\varepsilon^{-2})$, as desired. The theorem follows. □

4.5 Construction of the $(\delta, M_{\max}, M_{\min}, \widehat{\mathcal{P}})$-ADSP Data Structure

In this section we describe a step-by-step construction of the $(\delta, M_{\max}, M_{\min}, \widehat{\mathcal{P}})$-ADSP data structure (as defined in Definition 4.3.3) whose performance is described by Theorem 4.3.4. To this end, let us fix our set $\widehat{\mathcal{P}} := \bigcup_{j=1}^{\lceil \log n \rceil} \mathcal{P}(S_j, 2^j)$, where $\{S_j\}_j$ are the sampled subsets of vertices as in Definition 4.3.1. Note that for each j the expected size of S_j is $O(n \log n/2^j)$.

For given length vector $\boldsymbol{l}$, and real number ρ, let us define $\boldsymbol{l}^{[\rho]}$ to be a length vector with $l_e^{[\rho]} = \lceil l_e/\rho \rceil \rho$ i.e. $\boldsymbol{l}^{[\rho]}$ corresponds to rounding-up the lengths of the arcs given by $\boldsymbol{l}$ to the nearest multiple of ρ. The key relation between $\boldsymbol{l}$ and $\boldsymbol{l}^{\rho}$ is captured by the following simple lemma.

Lemma 4.5.1. *For any $\rho > 0$, length vector $\boldsymbol{l}$, and $1 \leq j \leq \lceil \log n \rceil$, if there exists a path $p \in \mathcal{P}(S_j, 2^j)$ of length $l(p)$ then $l^{[\rho/2^j]}(p) \leq l(p) + \rho$.*

Proof. Consider a path $p \in \mathcal{P}(S_j, 2^j)$, for some j, by definition we have

$$l^{[\rho/2^j]}(p) = \sum_{e \in p} l_e^{[\rho/2^j]} \leq \sum_{e \in p} (l_e + \rho/2^j) \leq l(p) + \rho,$$

since p can have at most 2^j arcs. □

As suggested by the above lemma, the basic idea behind our $(\delta, M_{\max}, M_{\min}, \widehat{\mathcal{P}})$-ADSP data structure construction is to maintain for each j *exact* shortest paths from a set *larger* than $\mathcal{P}(S_j, 2^j)$ (namely, $\mathcal{P}(S_j)$), but with respect to the *rounded* version $\boldsymbol{l}^{[\delta M_{\min}/2^j]}$ of the lengths $\boldsymbol{l}$, and to cap the length of these paths at $M_{\max} + \delta M_{\min}$. Note that we do not require our $(\delta, M_{\max}, M_{\min}, \widehat{\mathcal{P}})$-ADSP data structure to output paths from $\widehat{\mathcal{P}}$, thus this approach yields a correct solution. Moreover, as we show in section 4.5.1, using existing tools from dynamic graph algorithms we can obtain an implementation of this approach whose performance is close to the one postulated by Theorem 4.3.4, but with *linear* dependence of the maintenance cost on the ratio $\frac{M_{\max}}{M_{\min}}$ (as opposed to logarithmic one), and rather high service cost of $\mathsf{Increase}(\cdot,\cdot)$ requests. We alleviate these shortcomings in section 4.5.2, where we also prove Theorem 4.3.4.

4.5.1 Implementation of the $(\delta, M_{\max}, M_{\min}, \widehat{\mathcal{P}})$-ADSP with Linear Dependence on $\frac{M_{\max}}{M_{\min}}$

An important feature of the rounded length vector $\boldsymbol{l}^{[\rho]}$ for any $\rho > 0$, is that after dividing it by ρ we obtain a length vector that assigns integral lengths to arcs. Therefore, we are able to take advantage of existing tools for solving decremental shortest path problem in dynamic graphs with integer arc lengths. We start by defining this problem formally.

Definition 4.5.2. *For any integer $r \geq 0$ and a set of paths $\mathcal{Q} \subseteq \mathcal{P}$, let the* decremental $(r, \mathcal{Q})$-shortest path problem *($(r, \mathcal{Q})$-DSPP for short) be a problem in which one maintains a directed graph G with positive* integral *weights on its arcs, and that supports four operations:*

- $\overline{\mathsf{Distance}}(u, v)$*, for $u, v \in V$: returns the length of the shortest u-v path in $\mathcal{Q}$ if this length is at most r, and ∞ otherwise.*
- $\overline{\mathsf{Increase}}(e, t)$*, for $e \in E$ and integer $t \geq 0$: increases the length of the arc e by t*
- $\overline{\mathsf{Path}}(u, v)$*, for $u, v \in V$: returns a u-v path of length $\overline{\mathsf{Distance}}(u, v)$, as long as $\overline{\mathsf{Distance}}(u, v) \neq \infty$.*
- $\overline{\mathsf{SSrcDist}}(u)$*, for $u \in V$: returns $\overline{\mathsf{Distance}}(u, v)$ for all $v \in V$.*

We state first the following lemma which is just a simple and known extension of the classical construction of Even and Shiloach [60] (see also [120]).

Lemma 4.5.3. *For any $s \in V$ and positive integer r, $(r, \mathcal{P}(\{s\}))$-DSPP data structure can be maintained in total time $\widetilde{O}(mr)$ plus additional $O(\log n)$ per each $\overline{\mathsf{Increase}}(\cdot, \cdot)$ request. Each $\overline{\mathsf{Distance}}(\cdot, \cdot)$ request can be answered in $O(1)$ time and each $\overline{\mathsf{Path}}(\cdot, \cdot)$, and $\overline{\mathsf{SSrcDist}}(\cdot)$ query - in $O(n)$ time.*

Proof. First, we notice that to prove the lemma it is sufficient to show that we can maintain, within desired time bounds, the single-source shortest paths tree of all v-s paths that have length at most r, for any $v \in V$. Indeed, once we achieve this, it will also imply that we can keep the single-source shortest paths tree of all s-v paths having length at most r, for any $v \in V$. Now, to implement $(r, \mathcal{P}(\{s\}))$-DSPP data structure we just maintain both single-source shortest path trees and whenever queried $\overline{\mathsf{Distance}}(u, v)$ we answer $\overline{\mathsf{Distance}}(u, s) + \overline{\mathsf{Distance}}(s, v)$ if this sum does not exceed r and (u, v) is a source sink pair (i.e. the corresponding u-v path is in $\mathcal{P}$); and ∞ otherwise. Note that our trees allow finding $\overline{\mathsf{Distance}}(u, s)$ and $\overline{\mathsf{Distance}}(s, u)$ in $O(1)$ time. Similarly, we can answer request $\overline{\mathsf{Path}}(u, v)$, as long as $\overline{\mathsf{Distance}}(u, s) + \overline{\mathsf{Distance}}(s, v) \leq r$, by just concatenating paths $\overline{\mathsf{Path}}(u, s)$ and $\overline{\mathsf{Path}}(s, v)$ that we can obtain from our trees in $O(n)$ time. Finally to answer $\overline{\mathsf{SSrcDist}}(u)$ request we just issue a $\overline{\mathsf{Distance}}(u, v)$ request – that we already know how to handle – for each $v \in V$. It is easy to see that all the running times will fit within the desired bounds as long as our maintenance of single-source shortest paths tree will obey these bounds.

Our way of maintaining such a single-source shortest path tree of all v-s paths is a simple extension of the classical construction of Even and Shiloach [60] (see also [120]) who showed how to dynamically maintain in an *unweighted* directed graph, a decremental single-source shortest paths tree up to depth d, in $O(md)$ total running time.

In our data structure, each vertex v will keep a variable $d[v]$ whose value will be always equal to v's current distance to s. Moreover, each vertex v keeps a priority queue $Q[v]$ of all its outgoing arcs, where the key of a particular arc (v,u) is equal to the current value of $d[u] + l_{(v,u)}$. We want the queue to support three operations: $Insert(e,t)$ that adds an arc to a queue with key equal to t, $FindMin$ - returns the arc with smallest value of the key, and $SetKey(e,t)$ that sets the value of the key of arc e to t. By using e.g. Fibonacci heap implementation of such queue, $FindMin$ can be performed in $O(1)$ time, and each of the remaining operations can be done in $O(\log n)$ amortized time.[7]

The initialization of the data structure can be easily done by computing the single-source shortest path tree from s using Dijkstra algorithm and inserting arcs into appropriate queues, which takes $O(m \log n)$ time. Also, $\overline{\mathsf{Distance}}(\cdot, s)$ requests can be easily answered in $O(1)$ time by just returning $d[v]$. Finally, the implementations of the $\overline{\mathsf{Path}}(\cdot, s)$, and $\overline{\mathsf{Increase}}(\cdot,\cdot)$ can be found in Figure 4-6. Clearly, answering $\overline{\mathsf{Path}}(\cdot, s)$ query takes at most $O(n)$ time. Now, the total time needed to serve w $\overline{\mathsf{Increase}}(\cdot,\cdot)$ request is at most $O(\log n)$ times the total number of $Scan(\cdot)$ calls. But, since for a particular arc $e = (u,v)$ $Scan(e)$ is called only if either $\overline{\mathsf{Increase}}(e,\cdot)$ was called; or $d[v] < \infty$ and $d[v]$ increases by at least one, we see that this number is at most $m(r+1) + w$, which gives the desired running time.

□

We combine now the above construction of $(r, \mathcal{P}(\{s\}))$-DSPP data structure to obtain an implementation of $(r, \mathcal{P}(U))$-DSPP.

Lemma 4.5.4. *For any $S \subseteq V$ and positive integer r, $(r, \mathcal{P}(S))$-DSPP data structure can be maintained in total time $\widetilde{O}(mr|S|)$ plus additional $\widetilde{O}(|S|)$ per each $\overline{\mathsf{Increase}}(\cdot,\cdot)$ request. Each $\overline{\mathsf{Distance}}(\cdot,\cdot)$ query can be answered in $O(|S|)$ time, each $\overline{\mathsf{Path}}(\cdot,\cdot)$ – in time $O(n)$, and each $\overline{\mathsf{SSrcDist}}(\cdot)$ query - in $O(m + n\log n)$ time.*

Proof. We maintain $(r, \mathcal{P}(\{s\}))$-DSPP data structure R_s as in Lemma 4.5.3 for each $s \in S$. Clearly, the maintenance cost is $\widetilde{O}(mr|S|)$ plus additional $\widetilde{O}(|S|)$ per each $\overline{\mathsf{Increase}}(\cdot,\cdot)$ operation – we just forward each $\overline{\mathsf{Increase}}(\cdot,\cdot)$ operation to each R_s. Now, to serve $\overline{\mathsf{Distance}}(u,v)$ request we just issue $\overline{\mathsf{Distance}}(u,v)$ query to each R_s that we maintain and return the answer yielding minimal value. Answering $\overline{\mathsf{Path}}(u,v)$ request consist of just querying each R_s with $\overline{\mathsf{Distance}}(u,v)$, and forwarding $\overline{\mathsf{Path}}(u,v)$ request to R_s returning the minimal distance. Finally, to serve $\overline{\mathsf{SSrcDist}}(u)$ query, we construct a graph $G_{u,S}$ that consists of G equipped with current length vector $\boldsymbol{l}$, and additional vertex u' from which there is arc to each $s \in S$ with length corresponding to the

[7]Note that $SetKey(e,t)$ operation can be translated to 'standard' priority queue operations as: decrease the key of e by a value of ∞, extract the minimal element, and insert e again with new value of the key.

```
Procedure Path(v, s):
  if v = s then
  |   return an empty path ∅
  else
  |   e = (v, u) ← Q[v].FindMin
  |   return Path(u, s) ∪ {e}
  end

Procedure Increase(e, t):
  l_e ← l_e + t
  Scan(e)

Procedure Scan(e) :
  Q[u].SetKey(e, d[v] + l_e)
  f = (u, v') ← Q[u].FindMin
  if d[v'] + l_f > r then
  |   d[u] ← ∞
  end
  if d[v'] + l_f > d[u] then
  |   d[u] ← d[v'] + l_f
  |   foreach arc f' incoming to u do Scan(f')
  end
```

Figure 4-6: Implementation of procedures $\overline{\text{Path}}(v, s)$ and $\overline{\text{Increase}}(e, t)$, where $e = (u, v)$

distance from u to s with respect to $\boldsymbol{l}$. Note that construction of the graph $G_{u,S}$ can be performed in $O(m)$ time – in particular the length of each arc (u', s) can be obtained by querying R_s with $\overline{\mathsf{Distance}}(u, s)$. It is easy to see that if we compute single-source shortest path distances in $G_{u,S}$ from u' to all $v \in V$ using Dijkstra's algorithm then we can obtain the value of $\overline{\mathsf{Distance}}(u, v)$ for each $v \in V$ by just returning the computed value if it is at most r and (u, v) is a source-sink pair (so, the corresponding u-v path is in $\mathcal{P}$); and returning ∞ otherwise. Obviously, the total time required is $O(m + n \log n)$, as desired. □

We proceed now to designing an implementation of $(\delta, M_{\max}, M_{\min}, \widehat{\mathcal{P}})$-ADSP data structure that meets the time bounds of Theorem 4.3.4 except it maintenance time has linear – instead of logarithmic – dependence on $\frac{M_{\max}}{M_{\min}}$, and the time needed to serve $\mathsf{Increase}(\cdot, \cdot)$ request is much larger.

Lemma 4.5.5. *For any $\delta > 0$,and any $M_{\max}, M_{\min}$ such that $M_{\max} > 2M_{\min} > 0$, we can maintain $(\delta, M_{\max}, M_{\min}, \widehat{\mathcal{P}})$-ADSP data structure in total expected time $\widetilde{O}(mn\frac{M_{\max}}{\delta M_{\min}})$ plus additional $\widetilde{O}(n)$ per each $\mathsf{Increase}(\cdot, \cdot)$ request. Each $\mathsf{Distance}(\cdot, \cdot, \cdot)$ and $\mathsf{Path}(\cdot, \cdot, \cdot)$ query can be answered in $\widetilde{O}(n)$ time, and each $\mathsf{SSrcDist}(\cdot, \cdot)$ query - in $\widetilde{O}(m)$ time.*

Proof. Let $\boldsymbol{l}$ be the lengths of our graph G. For each $1 \leq j \leq \lceil \log n \rceil$ we maintain a $(\lceil M_j/\rho_j \rceil, \mathcal{P}(S_j))$-DSPP data structure R_j with respect to lengths $\boldsymbol{l}^j := l^{[\rho_j]}/\rho_j$, where $\rho_j := \delta M_{\min}/2^j$, and $M_j := M_{\max} + \delta M_{\min}$. Note that, by definition, $\boldsymbol{l}^j$ is integral, so we are allowed to use the data structure from Lemma 4.5.4. Moreover, by Lemma 4.5.4 and Lemma 4.5.1 applied with $\rho = \rho_j 2^j$, we see that if there is an s-t path p in $\mathcal{P}(S_j, 2^j)$ of length $l(p) \in [M_{\min}, M_{\max}]$ then R_j maintains a s-t path $p' \in \mathcal{P}(S_j)$ whose length with respect to $\boldsymbol{l}$ is at most $l(p) + 2^j \rho_j = l(p) + \delta M_{\min}$.

Now, to answer a $\mathsf{Distance}(u, v, \beta)$ query we just issue an $\overline{\mathsf{Distance}}(u, v)$ query to all R_j and return the value that is minimal after multiplying it by the respective value of ρ_j. Note that we are ignoring the value of β here - since $\beta \geq M_{\min}$, the accuracy of our answers is still sufficient. Similarly, as long as $\mathsf{Distance}(u, v, \beta) \neq \infty$, we answer $\mathsf{Path}(u, v, \beta)$ query by just returning the result of $\overline{\mathsf{Path}}(u, v)$ query forwarded to R_j whose $\overline{\mathsf{Distance}}(u, v)$ (after multiplying by ρ_j) is minimal. We implement answering $\mathsf{SSrcDist}(u, \beta)$ query by just issuing $\overline{\mathsf{SSrcDist}}(u)$ queries to each R_j, and for each $v \in V$ we return the reported distance that is minimal (after multiplication by respective ρ_j). Finally, whenever there is a $\mathsf{Increase}(e, \omega)$ request, we increase the length function l accordingly and issue $\overline{\mathsf{Increase}}(e, \lceil (l_e + \omega)/\sigma_j \rceil - \lceil l_e/\sigma_j \rceil)$ request to each R_j.

To analyze the performance of this implementation, we note that by Lemma 4.5.4 each $\mathsf{Distance}(\cdot, \cdot, \cdot)$ requires $O(\sum_j |S_j|) = \widetilde{O}(n)$ time, each $\mathsf{Path}(\cdot, \cdot, \cdot)$ – $\widetilde{O}(n)$ time, and each $\mathsf{SSrcDist}(\cdot, \cdot)$ – $\widetilde{O}(m)$ time. Also, the cost of serving $\mathsf{Increase}(e, \omega)$ request is $\widetilde{O}(\sum_j |S_j|) = \widetilde{O}(n)$. As a result, the total expected maintenance cost is, by Lemma 4.5.4:

$$\widetilde{O}(\sum_{j=1}^{\lceil \log n \rceil} mE[|S_j|]\lceil M_j/\rho_j \rceil) = \widetilde{O}(mn \sum_{j=1}^{\lceil \log n \rceil} \frac{2^j M_{\max}}{\delta M_{\min} 2^j}) = \widetilde{O}(mn\frac{M_{\max}}{\delta M_{\min}}),$$

where we used the fact that expected size of S_j is $O(n \log n/2^j)$. The lemma follows. □

4.5.2 Proof of Theorem 4.3.4

As we mentioned in section 4.3, in our applications the ratio of $M_{\max}$ to $M_{\min}$ can be very large i.e. $\Omega(n^{1/\varepsilon})$ for $\varepsilon < 1/2$. Therefore, the linear dependence on this ratio of the maintenance time of the $(\delta, M_{\max}, M_{\min}, \widehat{\mathcal{P}})$-ADSP data structure from Lemma 4.5.5 is still prohibitive. To address this issue we refine our construction in the following lemma. Subsequently, we will deal with large service time of $\mathsf{Increase}(\cdot,\cdot)$ requests in the proof of Theorem 4.3.4.

Lemma 4.5.6. *For any $\delta > 0$, $M_{\max} \geq 2M_{\min} > 0$, $(\delta, M_{\max}, M_{\min}, \widehat{\mathcal{P}})$-ADSP data structure can be maintained in total expected time $\widetilde{O}(mn\frac{\log M_{\max}/M_{\min}}{\delta})$ plus additional $\widetilde{O}(n \log \frac{1}{\delta})$ per each $\mathsf{Increase}(\cdot,\cdot)$ request in the processed sequence. Each $\mathsf{Distance}(\cdot,\cdot,\cdot)$ and $\mathsf{Path}(\cdot,\cdot,\cdot)$ query can be answered in $\widetilde{O}(n)$ time, and each $\mathsf{SSrcDist}(\cdot,\cdot)$ query – in $\widetilde{O}(m)$ time.*

Proof. For each $0 \leq b \leq \lfloor \log \frac{M_{\max}}{M_{\min}} \rfloor$, let us define $M^b_{\min} := 2^b M_{\min}$, and $M^b_{\max} = 2^{b+2} M_{\min}$. We will maintain for each b, a $(\delta, M^b_{\max}, M^b_{\min}, \widehat{\mathcal{P}})$-ADSP data structure R_b as in Lemma 4.5.5. Intuitively, we divide the interval $[M_{\min}, M_{\max}]$ into exponentially growing and partially overlapping intervals $[M^b_{\min}, M^b_{\max}]$, and we will make each R_b responsible for queries with β falling into interval $[M^b_{\min}, M^b_{\max}/2]$.

More precisely, upon receiving $\mathsf{Distance}(u, v, \beta)$, $\mathsf{Path}(u, v, \beta)$, or $\mathsf{SSrcDist}(u, \beta)$ request, we just pass it to the unique R_b with $M^b_{\min} \leq \beta \leq M^b_{\max}/2$, and report back the obtained answer. By Lemma 4.5.5 and definition of $(\delta, M^b_{\max}, M^b_{\min}, \widehat{\mathcal{P}})$-ADSP data structure, the supplied answer is correct, and the service cost is within desired bounds. Also, the part of the total expected maintenance cost that is independent of the number of $\mathsf{Increase}(\cdot,\cdot)$ requests is at most $\widetilde{O}(mn\frac{\log M_{\max}/M_{\min}}{\delta})$, as needed.

Therefore, it remains to design our way of handling $\mathsf{Increase}(\cdot,\cdot)$ requests, and bound the corresponding service cost. A straight-forward approach is to update the length vector $\boldsymbol{l}$ accordingly upon receiving $\mathsf{Increase}(e, \omega)$ request, and forward this request to all R_b. This would, however, result in $\widetilde{O}(n \log \frac{M_{\max}}{M_{\min}})$ service cost which is slightly suboptimal from our point of view. Our more refined implementation of handling $\mathsf{Increase}(e, \omega)$ request is based on two observations. First, we note that if at any point of time the length of arc e increases to more than $2M^b_{\max}$ for some b, we can safely increase the length of this arc in R_b to ∞ without violating the correctness of the answers supplied by R_b – we call such event *deactivation of e in R_b*. Second, we don't need to forward $\mathsf{Increase}(e, \omega)$ requests to R_b for which $l_e + \omega < \sigma^b_{\lceil \log n \rceil}$, where $\sigma^b_j = \frac{\delta M^b_{\min}}{2^j}$, and l_e is the current length of the arc e. This is so, since the construction of R_b's from Lemma 4.5.5 will have the rounded length $l^{[\sigma^b_j]}(e)$ of e still equal to σ^b_j, for every j. Therefore, instead of passing to such R_b $\mathsf{Increase}(e, \omega)$ requests each time they are issued, we just send an $\mathsf{Increase}(e, \omega')$ request to R_b once the length of e

exceeds $\sigma^b_{\lceil \log n \rceil}$, where ω' is the total increase of the length of e from the beginning up to the current value of l_e – we call such an event *activation of e in R_b*.

In the light of the above, our handling of a $\mathsf{Increase}(e, \omega)$ request is as follows. Let b_- be the largest b with $2M^b_{\max} < l_e + \omega$, and let b_+ be the largest b with $l_e + \omega \geq \sigma^b_{\lceil \log n \rceil}$. We start by deactivating e in all R_b with $b \leq b_-$ in which e wasn't already deactivated, and activating e (by increasing the length of e to l_e) in all R_b with $b_- < b \leq b_+$ in which it wasn't activated yet. Next, we issue $\mathsf{Increase}(e, \omega)$ request to all R_b with $b_- < b \leq b_+$, and we increase l_e by ω. It is not hard to see that by the above two observations, this procedure does not violate the correctness of our implementation of $(\delta, M_{\max}, M_{\min}, \widehat{\mathcal{P}})$-ADSP data structure. Now, to bound the time needed to service $\mathsf{Increase}(\cdot, \cdot)$ request we note that each arc e can be activated and deactivated in each R_b at most once, and each such operation takes $\widetilde{O}(n)$ time. So, the total cost of these operation is at most $\widetilde{O}(mn \log \frac{M_{\max}}{M_{\min}})$ and this cost can be amortized within the total maintenance cost. To bound the time taken by processing the $\mathsf{Increase}(e, \omega)$ requests passed to all R_b with $b_- < b \leq b_+$, we note that $l_e + \omega < 2M^{b_-+1}_{\max} \leq 2^{j+3}\sigma_j^{b_-+1}/\delta$ for any j, thus $b_+ - b_- - 1 \leq \log \frac{2^{\lceil \log n \rceil + 3}}{\delta}$, and the total service time is at most $\widetilde{O}(n \log \frac{1}{\delta})$, as desired. The lemma follows.

□

We are ready to prove Theorem 4.3.4.

of Theorem 4.3.4. We maintain $(\delta/2, M_{\max}, M_{\min}, \widehat{\mathcal{P}})$-ADSP data structure R as in Lemma 4.5.6. While serving the sequence of requests, we pass to R all the $\mathsf{Path}(\cdot, \cdot, \cdot)$ requests and return back the answers supplied by R. Similarly, we pass $\mathsf{Distance}(\cdot, \cdot, \beta)$ and $\mathsf{SSrcDist}(\cdot, \beta)$ to R, and return back the values supplied by R with $\frac{\delta\beta}{2}$ added to each of them. Finally, in case of $\mathsf{Increase}(\cdot, \cdot)$ requests we pass them to R in an *aggregate* manner. Namely, in addition to $\boldsymbol{l}$ – the (evolving) length function of the graph G – we also maintain an *aggregated* length function $\hat{\boldsymbol{l}}$. Initially, $\hat{\boldsymbol{l}} = \mathbf{1}$, and later, as $\boldsymbol{l}$ evolves in an on-line manner, we increase $\hat{l}_e$ to l_e for given arc e, each time l_e becomes greater than $\max\{(1 + \delta/8)\hat{l}_e, \delta M_{\min}/4n\}$. Note that this definition ensures that we have always $\hat{l}_e \leq l_e \leq (1 + \delta/8)\hat{l}_e + \delta M_{\min}/4n$ for any arc e. Now, instead of passing to R an $\mathsf{Increase}(e, \omega)$ request each time it is issued, we only issue an $\mathsf{Increase}(e, l_e - \hat{l}_e)$ request to R each time the value of $\hat{l}_e$ increases to l_e. In other words, we make R to work with respect to the length vector $\hat{\boldsymbol{l}}$ instead of $\boldsymbol{l}$.

Clearly, by Lemma 4.5.6 the time needed to answer $\mathsf{Distance}(\cdot, \cdot, \cdot)$, $\mathsf{Path}(\cdot, \cdot, \cdot)$, and $\mathsf{SSrcDist}(\cdot, \cdot)$ queries is within our intended bounds. Also, we can raise the length l_e of an arc e to ∞ once its length exceeds $2M_{\max}$, without violating the correctness of our implementation of $(\delta, M_{\max}, M_{\min}, \widehat{\mathcal{P}})$-ADSP data structure. Thus, for given arc e, the length $\hat{l}_e$ can increase at most $\lceil \log_{(1+\delta/8)} \frac{4M_{\max}n}{\delta M_{\min}} \rceil + 1 = \widetilde{O}(\frac{\log M_{\max}/M_{\min}}{\delta})$ times. Therefore, we issue at most $\widetilde{O}(m\frac{\log M_{\max}/M_{\min}}{\delta})$ $\mathsf{Increase}(\cdot, \cdot)$ requests to R, which by Lemma 4.5.6 takes at most $\widetilde{O}(mn\frac{\log M_{\max}/M_{\min}}{\delta})$ time to process. As a result, the total maintenance time of our construction is $\widetilde{O}(mn\frac{\log M_{\max}/M_{\min}}{\delta})$ plus additional $O(1)$ time per each $\mathsf{Increase}(\cdot, \cdot)$ request in the sequence, as desired.

To prove that our construction is a correct implementation of $(\delta, M_{\max}, M_{\min}, \widehat{\mathcal{P}})$-ADSP data structure, consider some s-t path p in $\widehat{\mathcal{P}}$ whose length $l(p)$ is at most 2β for some $\beta \in [M_{\min}, M_{\max}/2]$. Now, upon being queried with $\mathsf{Distance}(s,t,\beta)$ request, our data structure will return a value $d' = d + \delta\beta/2$, where d is the value returned by R as an answer to $\mathsf{Distance}(s,t,\beta)$ request passed. Since $\hat{l}(p) \leq l(p)$, we have $d' = d + \delta\beta/2 \leq \hat{l}(p) + \delta\beta/2 + \delta\beta/2 \leq l(p) + \delta\beta$, as desired. Moreover, upon $\mathsf{Path}(s,t,\beta)$ query we return a path p' with $\hat{l}(p') \leq d$. This means that

$$\begin{aligned} l(p') \leq \sum_{e \in p'} (1+\delta/8)\hat{l}_e + \delta M_{\min}/4n &\leq (1+\delta/8)\hat{l}(p') + \delta\beta/4 \\ &\leq (1+\delta/8)d + \delta\beta/4 \leq d + \delta\beta/2 = d', \end{aligned}$$

since $d \leq \hat{l}(p) + \delta\beta/2 \leq 2\beta + \delta\beta/2 < 3\beta$ for $\delta < 1$. The theorem follows.

□

4.6 Maximum Concurrent Flow Problem

Recall that in the maximum concurrent flow problem, in addition to the capacitated graph $G = (V, E, \boldsymbol{u})$ (once again, we assume that $\min_e u_e = 1$, and define $U := \max_e u_e$), and k source-sink pairs $\{(s_i, t_i)\}_i$, we also have a demand $d_i > 0$ associated with each commodity i. The task is to find a feasible multicommodity flow routing θd_i units of each commodity i that maximizes the rate θ.

Let as take θ^* to denote, from now on, the maximum concurrent flow in G (with respect to the source-sink pairs $\{(s_i, t_i)\}_i$ and the corresponding demands $\{d_i\}_i$). We want to show first that we can assume without loss of generality that $1 \leq \theta^* \leq km$.[8] To this end, let us define $\eta_i := B_i/d_i$, where B_i is the maximum bottleneck of an s_i-t_i path. Note that using the algorithm of Fredman and Tarjan [68] we can compute all of η_is in $\widetilde{O}(mn)$ time. Let $\eta^* := \min_i \eta_i$. As we know that the value of maximum s_i-t_i flow can be at most $m\eta^*$, we have that $\theta^* \leq \eta^* m$. On the other hand, as the value of each maximum s_i-t_i flow is at least B_i and we can simultaneously route $1/k$ fraction of each of them, we have that $\theta^* \geq \eta^*/k$. Thus, by scaling all the demands by η^*/k, we can reduce our consideration to $1 \leq \theta^* \leq mk$.

Similarly to the case of the maximum multicommodity flow in Section 4.2, we proceed to developing a multiplicative-weights-update-based framework for computing $(1-\varepsilon)$-approximation to the maximum concurrent flow problem. However, in contrast to the approach used in Section 4.2, our framework will not employ the flow packing machinery set forth in Section 2.8. It will use instead the original framework developed by Garg and Könemann [70]. This is motivated by the fact that even though one could employ the flow packing approach of Section 2.8 to the setting of maximum concurrent flow problem pretty easily, doing so in a way that yields the

[8] This observation was first made by Fleischer [65] and led to improvement of the running time of the original algorithm of Garg and Könemann [70] as they were using less efficient procedure for this.

desired performance is a bit involved. Therefore, for the sake of the exposition, in what follows we stick to the original approach of Garg and Könemann. We want to emphasis the fact that the dynamic-graph-based tools we developed in this chapter, could be applied to either of these approaches.

The algorithm of Garg and Könemann that $(1-O(\varepsilon))$-approximates the maximum concurrent flow problem works as follows. It starts with a zero flow – $f = \emptyset$, and setups for each arc e an initial length $l_e = \frac{\gamma}{u_e}$, where $\gamma = (\frac{m}{1-\varepsilon})^{-1/\varepsilon}$. The algorithm proceeds in *phases* – each one of them consists of k iterations. In each iteration i, the algorithm tries to route d_i units of flow from s_i to t_i. This is done by repeating the following steps. First, a shortest (with respect to current lengths $\boldsymbol{l}$) s_i-t_i path p is found. Next, the bottleneck capacity u, which is the minimum of the bottleneck capacity of the path p and the remaining demand $\widehat{d}_i$ to be routed, is computed. Then, the flow f is augmented by routing u units of commodity i along the path p. Finally, the length of each arc e of the path p is increased by a factor of $(1 + \frac{\varepsilon u}{u_e})$, and $\widehat{d}_i$ is decreased by u. The entire procedure stops when $D(\boldsymbol{l}) := \sum_e l_e u_e$ – the volume of G with respect to $\boldsymbol{l}$, is at least one. The summary of the algorithm can be found in Figure 4-7.

Input : Graph $G = (V, E, \boldsymbol{u})$, source-sink pairs $\{(s_i, t_i)\}_i$ with demands $\{d_i\}_i$, and an accuracy parameter $\varepsilon > 0$
Output: (Infeasible) flow f

Initialize $f \leftarrow \emptyset$, $l_e \leftarrow \frac{\gamma}{u_e}$ for all arcs $e \in E$, where $\gamma = (\frac{m}{1-\varepsilon})^{-1/\varepsilon}$
while $D(\boldsymbol{l}) < 1$ **do**
  for $i := 1, \ldots, k$ **do**
    $\widehat{d}_i \leftarrow d_i$
    while $D(\boldsymbol{l}) < 1$ and $\widehat{d}_i > 0$ **do**
      Find the shortest path p in $\mathcal{P}_i$
      Find the bottleneck capacity u of p $\quad (*\ u \leftarrow \min\{\widehat{d}_i, \min_{e \in p} u_e\}\ *)$
      $\widehat{d}_i \leftarrow \widehat{d}_i - u$
      Augment the flow f by routing u units of flow along the path p
      foreach arc e in p **do** $l_e \leftarrow l_e(1 + \frac{\varepsilon u}{u_e})$
    end
  end
end

Figure 4-7: Garg-Könemann algorithm for maximum concurrent flow.

Garg and Könemann prove the following lemmas.

Lemma 4.6.1. *If $\theta^* \geq 1$, the algorithm in Figure 4-7 terminates after at most $t := 1 + \theta^* \log_{1+\varepsilon} 1/\gamma = 1 + \frac{\theta^*}{\varepsilon} \log_{1+\varepsilon} \frac{m}{1-\varepsilon}$ phases.*

Lemma 4.6.2. *After $t-1$ phases, the algorithm has routed $(t-1)d_i$ units of each commodity i. Scaling the final flow by $\log_{1+\varepsilon} 1/\gamma$ yields a feasible flow that achieves a rate $\theta = \frac{t-1}{\log_{1+\varepsilon} 1/\gamma}$.*

Lemma 4.6.3. *If $\theta^* \geq 1$, then the final flow scaled down by $\log_{1+\varepsilon} 1/\gamma$ is feasible and has a value of at least $(1-3\varepsilon)\theta^*$.*

As we ensured that $1 \leq \theta^* \leq km$, it is easy to see that the above lemmas imply that the algorithm in Figure 4-7, after at most $1+\theta^* \log_{1+\varepsilon} 1/\gamma \leq 1+km \log_{1+\varepsilon} 1/\gamma$ phases, returns a $(1-3\varepsilon)$-approximation for the maximum concurrent flow. Unfortunately, the bound of $1 + mk \log_{1+\varepsilon} 1/\gamma$ on the number of phases is not sufficient to obtain the $\widetilde{O}((m+k)m\varepsilon^{-2})$ running time (in fact, it only establishes the time bound of $\widetilde{O}((m+k^2m)m\varepsilon^{-2})$). To reduce this dependence of the number of phases on θ^*, Garg and Könemann use a halving technique developed in [113]. They run the algorithm and if it does not stop after $T := 2\log_{1+\varepsilon} 1/\gamma + 1$ phases then, by Lemma 4.6.1, it must be that $\theta^* > 2$. In this case, they multiply the demands by two, so θ^* is halved and still at least one. Next, they continue the algorithm and keep doubling the demands again if it does not stop after T phases. Clearly, since initially $\theta^* \leq km$, after repeating such doubling at most $\log km$ times the algorithm stops, and thus the total number of phases is at most $T \log km$. The number of phases can be reduce further to $O(T)$ by first applying the above approach with constant ε to get a constant-factor approximation for θ^* – this takes $O(\log^2 km)$ phases – and then with at most $O(T)$ more phases the $(1-3\varepsilon)$-approximation is obtained. Having established this bound on the number of phases, the bound of $\widetilde{O}((m+k)m\varepsilon^{-2})$ on the running time of the whole algorithm follows easily.

Input : Graph $G = (V, E, \boldsymbol{u})$, source-sink pairs $\{(s_i, t_i)\}_i$ with demands $\{d_i\}_i$, and an accuracy parameter $\varepsilon > 0$
Output: (Infeasible) flow f

Initialize $f \leftarrow \emptyset$, and $l_e \leftarrow \frac{\gamma}{u_e}$ for all arcs $e \in E$, where $\gamma = (\frac{m}{1-\nu})^{-1/\nu}$, and $\nu := \varepsilon(1+\varepsilon)$
Sample sets $\{S_j\}_{j=1,\ldots,\lceil \log n \rceil}$ to indicate the set $\widehat{\mathcal{P}}$ (see Definition 4.3.1)
Initialize $(\varepsilon/2, 1, \gamma/U, \widehat{\mathcal{P}})$-ADSP data structure R as in Theorem 4.3.4
while $D(\boldsymbol{l}) < 1$ **do**
  for $i := 1, \ldots, k$ **do**
    $\widehat{d}_i \leftarrow d_i$
    while $D(\boldsymbol{l}) < 1$ and $\widehat{d}_i > 0$ **do**
      $p \leftarrow R$.Path(s_i, t_i, β), where β is guessed with binary search
      Find the bottleneck capacity u of p $\quad (*\ u \leftarrow \min\{\widehat{d}_i, \min_{e\in p} u_e\}\ *)$
      $\widehat{d}_i \leftarrow \widehat{d}_i - u$
      Augment the flow f by routing u units of flow along the path p
      foreach arc e in p **do** R.Increase$(e, \frac{\varepsilon u l_e}{u_e})$; $l_e \leftarrow l_e(1 + \frac{\varepsilon u}{u_e})$
    end
  end
end

Figure 4-8: Improved implementation of the algorithm from 4-7.

4.6.1 Our Algorithm

We present now a more efficient approximation scheme for maximum concurrent flow problem. Similarly, to the case of the maximum multicommodity flow problem, our improvement is based on making the algorithm from Figure 4-7 find the (approximately) shortest paths using $(\varepsilon, 1, \gamma/U, \widehat{\mathcal{P}})$-ADSP data structure instead of Dijkstra's algorithm. Our new implementation of procedure from Figure 4-7 is presented in Figure 4-8. Note that whenever the algorithm issues $\mathsf{Path}(u, v, \cdot)$ request it uses binary search to find the (u, v)-accurate β that yields small enough error – see discussion after Definition 4.3.3 for more details.

Let us define $\alpha(\boldsymbol{l}) := \sum_i d_i \mathrm{dist}_i(\boldsymbol{l})$, where $\mathrm{dist}_i(\boldsymbol{l})$ is the length of the shortest s_i-t_i path in $\widehat{\mathcal{P}}$ with respect to lengths $\boldsymbol{l}$. As it was the case for maximum multicommodity flow problem, we need to justify the fact that we focus our attention on shortest paths in $\widehat{\mathcal{P}}$ instead of the whole $\mathcal{P}$.

Lemma 4.6.4. *With high probability, for any length vector $\boldsymbol{l}$, $\alpha(\boldsymbol{l}) \leq D(\boldsymbol{l})/\theta^*$, where $D(\boldsymbol{l}) := \sum_e l_e u_e$.*

Proof. Similarly to the proof of Lemma 4.4.1, let us fix the optimal solution $f^* = (f_1^*, \ldots, f_k^*)$ that achieves the ratio of θ^*. By Lemma 4.3.2, with high probability, each p_j belongs to $\widehat{\mathcal{P}} \cap \mathcal{P}_{i(j)}$, where $p_1, \ldots, p_q$ are the flowpaths of f^* and $i(j)$ is the corresponding commodity. Now, since f^* obeys capacity constraints we have that

$$D(\boldsymbol{l}) = \sum_e l_e u_e \geq \sum_j l(p_j) f^*_{i(j)}(p_j) \geq \sum_{i=1}^{k} \theta^* d_i \min_{p \in \mathcal{P}_i} l(p) \geq \theta^* \alpha(\boldsymbol{l})$$

and the lemma follows. □

We are ready to prove the following theorem.

Theorem 4.6.5. *For any $0.15 > \varepsilon > 0$, there exists a Monte Carlo algorithm that finds a $(1-4\varepsilon)$-approximate solution for maximum concurrent flow problem in expected time $\widetilde{O}((m+k)n\varepsilon^{-2} \log U)$.*

Proof. As it was already mentioned, we will be using the algorithm of Garg and Könemann, in which we use the procedure from Figure 4-8 instead of the one presented in Figure 4-7. To bound the running time of this new algorithm, we recall that the preprocessing that ensures that $1 \leq \theta^* \leq km$ can be performed in $\widetilde{O}(mn)$ time. Also, we use the halving technique to ensure that the running time of the rest of the algorithm is bounded by the time needed to execute $O(T) = O(\log_{1+\varepsilon} 1/\gamma)$ phases of the procedure in Figure 4-8.

Now, it is easy to see that this running time is dominated by the cost of maintaining the data structure R, and the time needed to answer the requests $\mathsf{Path}(\cdot, \cdot, \cdot)$ (note there is at most $O(n)$ $\mathsf{Increase}(\cdot, \cdot)$ requests per each $\mathsf{Path}(\cdot, \cdot, \cdot)$ one). By Theorem 4.3.4 we know that the maintenance cost is $\widetilde{O}(mn\frac{\log U/\gamma}{\varepsilon}) = \widetilde{O}(mn\varepsilon^{-1}(\varepsilon^{-1} + \log U)) = \widetilde{O}(mn\varepsilon^{-2} \log U)$. To upperbound the time needed to answer $\mathsf{Path}(\cdot, \cdot, \cdot)$ requests, we

note that each such request results in augmentation of the flow, and each augmentation of the flow results either in increasing the length of at least one arc by $(1+\varepsilon)$, or it is the last augmentation for given commodity. But no arc can have length bigger than $(1+\varepsilon)/u_e$, because $l_e u_e \leq D(l)$ and we stop when $D(l) \geq 1$, thus the total number of augmentations of the flow is at most $m\lfloor \log_{(1+\varepsilon)} \frac{(1+\varepsilon)}{\gamma} \rfloor + k \cdot O(T) = \widetilde{O}(m\varepsilon^{-2} + kT)$, which results in $\widetilde{O}((m+k)n\varepsilon^{-2} \log\log U/\gamma) = \widetilde{O}((m+k)n\varepsilon^{-2} \log\log U)$ needed to answer all $\mathsf{Path}(\cdot,\cdot,\cdot)$ requests (including the time needed to perform each binary search for β). As a result, the algorithm works in $\widetilde{O}((m+k)n\varepsilon^{-2} \log U)$ time, as desired.

We proceed now to lowerbounding the value of the final flow f computed by our algorithm after scaling it down by the maximal congestion of the arcs. To this end, let, for given j, and $1 \leq q \leq q_j$, $f_{j,q}$ be the flow of commodity $i(j,q)$ that was routed along $s_{i(j,q)}$-$t_{i(j,q)}$ path $p_{j,q}$ in q-th augmentation of the flow f in phase j, where q_j is the total number of flow augmentations during phase j. Let $\boldsymbol{l}_{j,q}$ be the length vector $\boldsymbol{l}$ after routing the flow $f_{j,q}$. For any j and $q \geq 1$, the fact that we always find the $(s_{i(j,q)}, t_{i(j,q)})$-accurate β for the $\mathsf{Path}(s_{i(j,q)}, t_{i(j,q)}, \beta)$ query, implies that $l_{j,q-1}(p_{j,q}) \leq (1+\varepsilon)\mathrm{dist}_{i(j,q)}(\boldsymbol{l}_{j,q-1})$. Therefore, we have that

$$D(\boldsymbol{l}_{j,q}) \leq D(\boldsymbol{l}_{j,q-1}) + \nu \mathrm{dist}_{i(j,q)}(\boldsymbol{l}_{j,q-1}),$$

where $\nu := \varepsilon(1+\varepsilon)$.

As a result, since the lengths of arcs can only increase, and in each completed phase we route exactly d_i units of commodity i, for all $j \geq 1$ we have that

$$D(j) \leq D(j-1) + \nu \sum_i d_i \mathrm{dist}_i(\boldsymbol{l}_{j,q_j}) = D(j-1) + \nu\alpha(\boldsymbol{l}_{j,q_j}),$$

where $D(j)$ denotes $D(\boldsymbol{l}_{j,q_j})$.

By Lemma 4.6.4 we know that with high probability $\frac{D(j)}{\alpha(l_{j,q_j})} \geq \theta^*$ for all j, thus we have that

$$D(j) \leq \frac{D(j-1)}{1-\nu/\theta^*}.$$

But, $D(0) = \sum_e \frac{\gamma}{u_e)} u_e) = m\gamma$, so for $j \geq 1$

$$\begin{aligned} D(j) &\leq \frac{m\gamma}{(1-\nu/\theta^*)^j} \\ &= \frac{m\gamma}{(1-\nu/\theta^*)}(1 + \frac{\nu}{(\theta^*-\nu)})^{j-1} \\ &\leq \frac{m\gamma}{(1-\nu/\theta^*)} e^{\frac{\nu(j-1)}{\theta^*-\nu}} \\ &\leq \frac{m\gamma}{(1-\nu)} e^{\frac{\nu(j-1)}{\theta^*(1-\nu)}}, \end{aligned}$$

where the last inequality uses the fact that $\theta^* \geq 1$.

The algorithm stops at the first phase j_f during which $D(\boldsymbol{l}) \geq 1$. Thus,

$$1 \leq D(j_f) \leq \frac{m\gamma}{(1-\nu)} e^{\frac{\nu(j_f-1)}{\theta^*(1-\nu)}},$$

which in turn implies

$$j_f - 1 \geq \frac{\theta^*(1-\nu)\ln\frac{1-\nu}{m\gamma}}{\nu}$$

Since we had $j_f - 1$ successfully completed phases, the flow produced by our procedure routes at least $(j_f - 1)d_i$ units of each commodity i. Unfortunately, this flow may be not feasible – it may violate some capacity constraints. But in our algorithm the length l_e of any arc e cannot be bigger than $(1+\varepsilon)/u_e$. Thus, the fact that each arc starts with length $l_e := \gamma/u_e$ and each time a full unit of flow is routed through it, its length increases by a factor of at least $(1+\varepsilon)$, implies that the congestion incurred at e can be at most $\lfloor \log_{1+\varepsilon}(1+\varepsilon)/\gamma \rfloor$. Therefore, we see that the rate θ achieved by the final flow after scaling it down is at least

$$\theta \geq \frac{j_f - 1}{\lfloor \log_{1+\varepsilon}(1+\varepsilon)/\gamma \rfloor} \geq \frac{\theta^*(1-\nu)\ln\frac{1-\nu}{m\gamma}}{\nu \log_{1+\varepsilon} 1/\gamma}.$$

Plugging in $\gamma = (m/(1-\nu))^{-1/\nu}$ and unwinding the definition of ν yields

$$\theta \geq \frac{(1-\varepsilon(1+\varepsilon))^2 \ln(1+\varepsilon)}{\varepsilon(1+\varepsilon)}\theta^* \geq (1-4\varepsilon)\theta^*,$$

which proves that the flow is indeed a $(1-4\varepsilon)$-approximation of the maximum concurrent flow.

□

Chapter 5

(Multi-)Cut-Based Minimization Problems in Undirected Graphs

The focus of this chapter are the (multi-)cut-based minimization problems. This class of problems captures a variety of basic graph optimization tasks, including the minimum cut problem, the minimum s-t cut problem, the (generalized) sparsest cut problem, the balanced separator problem, minimum multi-cut problem, minimum multi-way-cut problem, and many other graph partitioning primitives. We present a general method of designing fast approximation algorithms for this class of problems in undirected graphs. In particular, we develop a technique that given any such problem that can be approximated quickly on trees, allows us to approximate it almost as quickly on general graphs while only losing a poly-logarithmic factor in the approximation guarantee.

To illustrate the applicability of this paradigm, we consider the undirected generalized sparsest cut problem and the balanced separator problem. By a simple use of the developed framework, we obtain the first poly-logarithmic approximation algorithms for these problems that run in time close to linear.

The main tool behind the results of this chapter is an efficient procedure that decomposes general graphs into simpler ones while approximately preserving the cut structure. This decomposition is inspired by the graph decomposition technique of Räcke that was developed in the context of oblivious routing schemes, and it employs some of the tools described in Section 2.6, as well as, draws upon some high-level ideas underlying the iterative approach of Spielman and Teng to solving Laplacian systems (cf. Section 2.7).

5.1 Introduction

(Multi-)cut-based graph problems are ubiquitous in optimization. They have been extensively studied – both from theoretical and applied perspective – in the context of flow problems, theory of Markov chains, geometric embeddings, clustering, community detection, VLSI layout design, and as a basic primitive for divide-and-conquer approaches (cf. [124]). For the sake of concreteness, we define an optimiza-

tion problem $\mathcal{P}$ to be an *(undirected) multi-cut-based (minimization) problem* if its every instance $P \in \mathcal{P}$ can be cast as a task of finding – for a given input undirected capacitated graph $G = (V, E, \boldsymbol{u})$ – a partition $\vec{C}^* := \{C_i^*\}_{i=1}^{k^*}$ of vertices of V such that:

$$\vec{C}^* = \text{argmin}_{\vec{C} \text{is a partition of} V} u(\vec{C}) f_P(\vec{C}), \tag{5.1}$$

where $u(\vec{C})$ is the sum of capacities of all the edges of G whose endpoints are in different C_is and f_P is an arbitrary non-negative function of $\vec{C}$ that depends on P, but not on the edge set E (and the capacities $\boldsymbol{u}$) of the graph G. It is not hard to see that a large number of graph problems fits this definition. For instance, the minimum cut problem (cf. [80, 107, 90, 132, 88]) corresponds to taking $f_P(\vec{C})$ to be 1 if $\vec{C}$ partitions the graph into exactly two pieces (i.e., it corresponds to a cut), and $+\infty$ otherwise; and the minimum s-t cut problem (that was discussed in Chapter 3) corresponds to a further restricting of $f_P(\vec{C})$ by making it equal to 1 only if $\vec{C}$ corresponds to a cut that separates s and t, and being $+\infty$ otherwise. Also, to see an example of a multi-cut problem captured by this definition, one can see that the multi-way cut problem in which we need to find a minimum capacity set of edges that disconnects some set of terminals (cf. [49, 50, 47, 48, 89]), corresponds to taking $f_P(\vec{C})$ to be always equal to 1 if $\vec{C}$ has exactly one of terminals in each part of the partition, and $+\infty$ otherwise.

For the sake of simplicity, we will focus from now on only on the cut-based minimization problems – the problems in which the function $f_P(\vec{C})$ is equal to $+\infty$ whenever the partition $\vec{C}$ consists of more than two parts. These problems can be viewed as the ones that can be cast as a task of finding a cut C^* such that:

$$C^* = \text{argmin}_{\emptyset \neq C \subset V} u(C) f_P(C), \tag{5.2}$$

where $u(C)$ is the capacity of the cut C in G and f_P, once again, is a non-negative function that depends on P and C, but not on the set E of edges of G and their capacities.

Even though we develop the framework for this special case, everything easily transfer over to the general multi-cut-based setting. The reason behind this is that for any partition $\vec{C} = \{C_i\}_{i=1}^{k}$, we can express $u(\vec{C})$ as a linear combinations of the capacities of the cuts $\{C_i\}_i$. Namely, we have

$$u(\vec{C}) = \frac{1}{2} \sum_{i=1}^{k} u(C_i).$$

Therefore, by linearity of expectation, we can conclude that if G is our input graph then any graph that approximates (in expectation) the cuts of G also approximates (in expectation) its multi-cuts. As we will see, this statement alone suffices to transplant the framework to the multi-cut setting.

For illustrative purposes, we will focus in this chapter on two particular examples of cut-based minimization problems – fundamental graph partitioning primitives – generalized sparsest cut problem and balanced separator problem.

In the *generalized sparsest cut* problem, we are given a graph $G = (V, E, \boldsymbol{u})$ and a demand graph $D = (V, E_D, \boldsymbol{d})$. Our task is to find a cut C^* that minimizes the *(generalized) sparsity* $u(C)/d(C)$ among all the cuts C of G. Here, $d(C)$ is the total demand $\sum_{e \in E_D(C)} d_e$ of the demand edges $E_D(C)$ of D cut by the cut C. Casted in our terminology, the problem is to find a cut C^* being $\operatorname{argmin}_C u(C) f_P(C)$ with $f_P(C) := 1/d(C)$.

An important problem that is related to the generalized sparsest cut problem is the *sparsest cut* problem. In this problem one aims at finding a cut C^* that minimizes the *sparsity* $u(C)/\min\{|C|, |\overline{C}|\}$ among all the cuts C of G. Note that if we considered an instance of generalized sparsest cut problem corresponding to the demand graph D being complete and each demand being $1/|V|$ (this special case is sometimes called the *uniform sparsest cut* problem) then the corresponding generalized sparsity of every cut C is within factor of two of its sparsity. Therefore, up to this constant factor, one can view the sparsest cut problem as a special case of the generalized sparsest cut problem.

On the other hand, the *balanced separator* problem corresponds to a task of finding a cut that minimizes the sparsity among all the *c-balanced cuts* C in G, i.e. among all the cuts C with $\min\{|C|, |\overline{C}|\} \geq c|V|$, for some specified constant $c > 0$ called the *balance constant* of the instance. In our framework, the function $f_P(C)$ corresponding to this problem is equal to $1/\min\{|C|, |\overline{C}|\}$ if $\min\{|C|, |\overline{C}|\} \geq c|V|$ and equal to $+\infty$ otherwise.

5.1.1 Previous Work on Graph Partitioning

The captured by the above-mentioned problems task of partitioning a graph into relatively large pieces while minimizing the number of edges cut is a fundamental combinatorial problem (see [124]) with a long history of theoretical and applied investigation. Since most of the graph partitioning problems – in particular, the ones we consider – are NP-hard, one needs to settle for approximation algorithms. In case of the sparsest cut problem and the balanced separator problem, there were two main types of approaches to approximating them.

First type of approach was based on spectral methods. Inspired by Cheeger's inequality [38] discovered in the context of Riemannian manifolds, Alon and Milman [9] transplanted it to discrete setting (cf. Section 2.5) and presented an algorithm for the sparsest cut problem that can be implemented to run in nearly-linear time (see Section 6 in [130]) by utilizing the fast Laplacian system solver (cf. Section 2.7). However, the approximation ratio of this algorithm depends on the conductance of the graph – being the conductance of the minimum-conductance cut (see equation (2.1) for definition of the conductance of a cut) – and can be as large as $\Omega(n)$ even in a graph with unit-capacity edges. Later, Spielman and Teng [129, 131] (see also [11] and [12] for a follow-up work) used this approach to design an algorithm for the balanced separator problem. However, both the running time and the approximation guarantee of this algorithms depend on the conductance of the graph, leading to an $\Omega(n)$ approximation ratio and large running time in the worst case.

The second type of approach relies on flow methods. It was started by Leighton

and Rao [101] who used linear programming relaxation of the maximum concurrent flow problem to establish the first poly-logarithmic approximation algorithms for many graph partitioning problems. In particular, an $O(\log n)$-approximation for the sparsest cut and the balanced separator problems. Both these algorithms can be implemented to run in $\widetilde{O}(n^2)$ time.

These spectral and flow-based approaches were combined in a seminal result of Arora, Rao, and Vazirani [17] to give an $O(\sqrt{\log n})$-approximation for the sparsest cut and the balanced separator problems.

In case of the generalized sparsest cut problem, the results of Linial, London, and Rabinovich [102], and of Aumann and Rabani [19] give a $O(\log r)$-approximation algorithms, where r is the number of vertices of the demand graph that are endpoints of some demand edge. Subsequently, Chawla, Gupta, and Räcke [37] extended the techniques from [17] to obtain an $O(\log^{3/4} r)$-approximation. This approximation ratio was later improved by Arora, Lee, and Naor [16] to $O(\sqrt{\log r}\log\log r)$.

5.1.2 Fast Approximation Algorithms for Graph Partitioning Problems

More recently, a lot of effort was put into designing algorithms for the graph partitioning problems that are very efficient while still having poly-logarithmic approximation guarantee. Arora, Hazan, and Kale [13] combined the concept of expander flows that was introduced in [17] together with multicommodity flow computations to obtain $O(\sqrt{\log n})$-approximation algorithms for the sparsest cut and the balanced separator problems that run in $\widetilde{O}(n^2)$ time.

Subsequently, Khandekar, Rao, and Vazirani [92] designed a primal-dual framework for graph partitioning algorithms and used it to achieve $O(\log^2 n)$-approximations for the same two problems running in $\widetilde{O}(m + n^{3/2})$ time needed to perform graph sparsification of Benczúr and Karger [26] followed by a poly-logarithmic number of maximum single-commodity flow computations. In [15], Arora and Kale introduced a general approach to approximately solving semi-definite programs which, in particular, led to $O(\log n)$-approximation algorithms for the sparsest cut and the balanced separator problems that also run in $\widetilde{O}(m + n^{3/2})$ time. Later, Orecchia, Schulman, Vazirani, and Vishnoi [110] obtained the same approximation ratio and running time as [15] by extending the framework from [92]. Recently, Sherman [123] presented an algorithm that for any $\varepsilon > 0$ works in $\widetilde{O}(m + n^{3/2+\varepsilon})$ time – corresponding to performing sparsification and $\widetilde{O}(n^{\varepsilon})$ maximum single-commodity flow computations – and achieves an approximation ratio of $O(\sqrt{\log n/\varepsilon})$. Therefore, for any fixed $\varepsilon > 0$, his algorithm has the best-known approximation guarantee while still having running time close to the time needed for (approximate) maximum s-t flow computation. By employing our maximum s-t flow algorithm from Chapter 3, the running time of Sherman's algorithm can be immediately improved to $\widetilde{O}(m + n^{4/3+\varepsilon})$.

Unfortunately, despite this progress in designing efficient poly-logarithmic approximation algorithms for the sparsest cut problem, if one is interested in the generalized sparsest cut problem then a folklore $O(\log r)$-approximation algorithm running

in $\widetilde{O}(n^2 \log U)$ time is still the fastest one known. This folklore result is obtained by combining an efficient implementation [26, 65] of the algorithm of Leighton and Rao [101] together with the results of Linial, London, and Rabinovich [102], and of Aumann and Rabani [19].

5.1.3 Overview of Our Results

In this chapter we present a general method of designing fast approximation algorithms for undirected (multi-)cut-based minimization problems. In particular, we design a procedure that given an integer parameter $k \geq 1$ and any undirected graph G with m edges, n vertices, and having integral edge capacities in the range $[1, \ldots, U]$, produces in $\widetilde{O}(m + 2^k n^{1+1/(2^k-1)} \log U)$ time a small number of trees $\{T_i\}_i$. These trees have a property that, with high probability, for any $\alpha \geq 1$, we can find an $(\alpha \log^{(1+o(1))k} n)$-approximation to given instance of *any* (multi-)cut-based minimization problem on G by just obtaining some α-optimal solution for each T_i and choosing the one among them that leads to the smallest objective value in G (see Theorem 5.4.1 for more details). As a consequence, we are able to transform any α-approximation algorithm for a (multi-)cut-based problem that works only on tree instances, to an $(\alpha \log^{(1+o(1))k} n)$-approximation algorithm for general graphs, while paying a computational overhead of $\widetilde{O}(m+2^k n^{1+1/(2^k-1)} \log U)$ time that, as k grows, quickly approaches close to linear time.

We illustrate the applicability of the above paradigm on two fundamental graph partitioning problems: the undirected (generalized) sparsest cut and the balanced separator problems. By a simple use of the framework to be presented in this chapter we obtain, for any integral $k \geq 1$, a $(\log^{(1+o(1))k} n)$-approximation algorithm for the generalized sparsest cut problem that runs in time $\widetilde{O}(m + |E_D| + 2^k n^{1+1/(2^k-1)} \log U)$, where $|E_D|$ is the number of the demand edges in the demand graph. Furthermore, in case of the sparsest cut and the balanced separator problems, we combine these techniques with the algorithm of Sherman [123] to obtain approximation algorithms that have even better dependence on k in the running time. Namely, for any $k \geq 1$ and $\varepsilon > 0$, we get a $(\log^{(1+o(1))(k+1/2)} n/\sqrt{\varepsilon})$-approximation algorithms[1] for these problems that run in time $\widetilde{O}(m + 2^k n^{1+1/(4 \cdot 2^k-1)+\varepsilon})$. We summarize the obtained results for these problems together with the previous work in Figure 5-1. One can see that even for small values of k, the running times of the obtained algorithms beat the multicommodity flow barrier of $\Omega(n^2)$ and the bound of $\Omega(m + n^{3/2})$ time corresponding to performing sparsification and single-commodity flow computation. Unfortunately, in the case of the generalized sparsest cut problem our approximation guarantee has poly-logarithmic dependence on n. This is in contrast to the previous algorithms for this problem that have their approximation guarantee depending on r.

[1] As was the case in all the previous work we described, we only obtain a pseudo-approximation for the balanced separator problem. Recall that an *α-pseudo-approximation* for the balanced separator problem with balance constant c is a c'-balanced cut C – for some other constant c' – such that C's sparsity is within α of the sparsest c-balanced cut. For the sake of convenience, in the rest of the chapter we don't differentiate between pseudo- and true approximation.

(Uniform) sparsest cut and balanced separator problem:

Algorithm	Approximation ratio	Running time
Alon-Milman [9]	$O(\Phi^{-1/2})$ ($\Omega(n)$ worst-case)	$\widetilde{O}(m/\Phi^2)$ ($\widetilde{O}(m)$ using [130])
Andersen-Peres [12]	$\widetilde{O}(\Phi^{-1/2})$ ($\widetilde{\Omega}(n)$ worst-case)	$\widetilde{O}(m/\Phi^{3/2})$
Leighton-Rao [101]	$O(\log n)$	$\widetilde{O}(n^2)$
Arora-Rao-Vazirani [17]	$O(\sqrt{\log n})$	polynomial time
Arora-Hazan-Kale [13]	$O(\sqrt{\log n})$	$\widetilde{O}(n^2)$
Khandekar-Rao--Vazirani [92]	$O(\log^2 n)$	$\widetilde{O}(m+n^{3/2})$
Arora-Kale [15]	$O(\log n)$	$\widetilde{O}(m+n^{3/2})$
Orecchia-Schulman--Vazirani-Vishnoi [110]	$O(\log n)$	$\widetilde{O}(m+n^{3/2})$
Sherman [123]	$O(\sqrt{\log n/\varepsilon})$	$\widetilde{O}(m+n^{3/2+\varepsilon})$ ($\widetilde{O}(m+n^{4/3+\varepsilon})$ using the algorithm from Chapter 3)
this thesis $k=1$	$(\log^{3/2+o(1)} n)/\sqrt{\varepsilon}$	$\widetilde{O}(m+n^{8/7+\varepsilon})$
this thesis $k=2$	$(\log^{5/2+o(1)} n)/\sqrt{\varepsilon}$	$\widetilde{O}(m+n^{16/15+\varepsilon})$
this thesis $k\geq 1$	$(\log^{(1+o(1))(k+1/2)} n)/\sqrt{\varepsilon}$	$\widetilde{O}(m+2^k n^{1+1/(4\cdot 2^k-1)+\varepsilon})$

Generalized sparsest cut problem:

Algorithm	Approximation ratio	Running time
Folklore ([101, 102, 19, 26, 65])	$O(\log r)$	$\widetilde{O}(n^2 \log U)$
Chawla-Gupta-Räcke [37]	$O(\log^{3/4} r)$	polynomial time
Arora-Lee-Naor [16]	$O(\sqrt{\log r}\log\log r)$	polynomial time
this thesis $k=2$	$\log^{2+o(1)} n$	$\widetilde{O}(m+\lvert E_D\rvert+n^{4/3}\log U)$
this thesis $k=3$	$\log^{3+o(1)} n$	$\widetilde{O}(m+\lvert E_D\rvert+n^{8/7}\log U)$
this thesis $k\geq 1$	$\log^{(1+o(1))k} n$	$\widetilde{O}(m+\lvert E_D\rvert+ +2^k n^{1+1/(2^k-1)}\log U)$

Figure 5-1: Here, n denotes the number of vertices of the input graph G, m the number of its edges, Φ its conductance, U is its capacity ratio, and r is the number of vertices of the demand graph $D=(V,E_D,d)$ that are endpoints of some demand edges. Also, $\widetilde{O}(\cdot)$ notation suppresses poly-logarithmic factors. The algorithm of Alon and Milman applies only to the sparsest cut problem.

5.1.4 Overview of the Techniques

The approach we take is inspired by the cut-based graph decomposition of Räcke [115] that was developed in the context of oblivious routing schemes (see [114, 21, 28, 81] for some previous work on this subject). This decomposition is a powerful tool in designing approximation algorithms for various undirected cut-based problem.[2] It is based on finding for a given graph G with n vertices and m edges, a convex combination $\{\lambda_i\}_i$ of decomposition trees $\{T_i\}_i$ such that: G is embeddable into each T_i, and this convex combination can be embedded into G with $O(\log n)$ congestion. One employs this decomposition by first approximating the desired cut-based problem on each tree T_i – this usually yields much better approximation ratio than general instances – and then extracting from obtained solutions a solution for the graph G while incurring only additional $O(\log n)$ factor in the approximation guarantee.

The key question motivating our results is: can the above paradigm be applied to obtain *very efficient* approximation algorithms for cut-based graph problems? After all, one could envision a generic approach to designing fast approximation algorithms for such problems in which one decomposes first an input graph G into a convex combination of structurally simple graphs G_i (e.g. trees), solves the problem quickly on each of these easy instances and then combines these solutions to obtain a solution for the graph G, while losing some (e.g. poly-logarithmic) factor in approximation guarantee as a price of this speed-up. Clearly, the viability of such scenario depends critically on how fast a suitable decomposition can be computed and how 'easy' are the graphs G_is from the point of view of the problems we want to solve.

We start investigation of this approach by noticing that if one is willing to settle for approximation algorithms that are of Monte Carlo-type then, given a decomposition of the graph G into a convex combination $\{(\lambda_i, G_i)\}_i$, one does not need to compute the solution for each of G_is to get the solution for G. It is sufficient to just solve the problem on a small number of G_is that are sampled from the distribution described by λ_is.

However, even after making this observation, one still needs to compute the decomposition of the graph to be able to sample from it and, unfortunately, Räcke's decomposition – that was aimed at obtaining algorithms that are just polynomial-time – has serious limitations when one is interested in time-efficiency. In particular, the running time of his decomposition procedure is dominated by $\widetilde{O}(m)$ all-pair shortest path computations and is thus prohibitively large from our point of view – cf. Section 5.3.1 for more details.

To circumvent this problem, we design an alternative and more general graph decomposition (cf. Theorem 5.3.6). Similarly to the case of Räcke's decomposition, our construction is based on combining the multiplicative-weights-updated-based routine (cf. Section 2.8 with the technique of embedding graph metrics into tree metrics. However – instead of the embedding result of Fakcharoenphol, Talwar, and Rao [62] that was used by Räcke – we employ the nearly-linear time algorithm of Abraham, Bartal, and Neiman [1] for finding low-average-stretch spanning trees (cf. Theorem

[2] As explained above, in the rest of the chapter, we will focus only on cut-based problems, but all the results we obtain hold also for the multi-cut-based setting.

2.6.3). This choice allows for a much more efficient implementation of our decomposition procedure at a cost of decreasing the quality of provided approximation from $O(\log n)$ to $\widetilde{O}(\log n)$. Also, even more crucially, inspired by the construction of the ultrasparsifiers of Spielman and Teng [129, 130] (see Section 2.7 for an overview of this construction), we allow flexibility in choosing the type of graphs into which our graph is decomposed. In this way, we are able to offer a trade-off between the structural simplicity of these graphs and the time needed to compute the corresponding decomposition.

Finally, by recursive use of our decomposition – together with sparsification technique – we are able to leverage the above-mentioned flexibility to design a procedure that can have its running time be arbitrarily close to nearly-linear and, informally speaking, allows us to sample from a decomposition of our input graph G into graphs whose structure is arbitrarily simple, but at a cost of getting proportionally worse quality of the reflection of the cut structure of G – see Theorem 5.3.7 for more details.

5.1.5 Outline of This Chapter

We start with some preliminaries in section 5.2. Next, in section 5.3, we introduce the key concepts of the chapter and state the main theorems. In section 5.4 we show how our framework leads to fast approximation algorithms for undirected cut-based minimization problems. We also construct there our algorithms for the (generalized) sparsest cut and balanced separator problems. Section 5.5 contains the description of our decomposition procedure that allows expressing general graphs as a convex combinations of simpler ones while approximately preserving cut-flow structure. We conclude in section 5.6 with showing how a recursive use of this decomposition procedure together with sparsification leads to the sampling procedure underlying our framework.

5.2 Preliminaries

The object of study in this chapter will be an undirected capacitated graphs $G = (V, E, \boldsymbol{u})$. By our convention, we will denote by U the *capacity ratio of* G being the maximum possible ratio between capacities of two edges of G i.e. $U := \max_{e,e' \in E} u(e)/u(e')$.

We will say, for some $V' \subseteq V$, that a subgraph $G' = (V', E', \boldsymbol{u}')$ is a *subgraph of* G *induced by* V' if E' consists of all the edges of G whose both endpoints are in V' and the capacity vector $\boldsymbol{u}'$ on these edges is inherited from the capacity vector $\boldsymbol{u}$ of G.

5.2.1 The Maximum Concurrent Flow Problem

The problem that will find useful in consideration in this chapter, is the maximum concurrent flow problem that we already studied in Chapter 4. For convenience, we recall its definition here – as in this chapter we are dealing only with undirected

graphs, we adjust its definition to this setting. In the maximum concurrent flow problem, we are given an undirected capacitated graph $G = (V, E, \boldsymbol{u})$, as well as, a set of k source sink pairs $\{(s_i, t_i)\}_i$ and corresponding positive demands $\{d_i\}_i$. The goal is to find a multicommodity flow $f = (f_1, \ldots, f_k)$, where each f_i is an s_i-t_i flow of value θd_i that is *feasible*, i.e. has $|f(e)| \leq u_e$ for each edge e, and maximizes the *rate* θ. Here, $|f(e)| := \sum_i |f^i(e)|$ is the total flow routed along the edge e by f.

For reasons that will be clear soon, in this chapter, we will describe the maximum concurrent flow instances in slightly different – although completely equivalent – way. Namely, we will represent the source-sink pairs $\{(s_i, t_i)\}_i$ and the demands $\{d_i\}_i$ via a *demand graph* $D = (V, E_D, \boldsymbol{d})$. In this graph, each source-sink pair (s_i, t_i) with demand d_i is represented by an edge $(s_i, t_i) \in E_D$ with its demand $d_{(s_i,t_i)} = d_i$. Clearly, this correspondence allows us not only represent a set of source-sink pairs as a demand graph, but also view any demand graph as a maximum concurrent flow problem instance.[3]

5.2.2 Embedability

A notion we will be dealing extensively with is the notion of a graph embedding.

Definition 5.2.1. *For given graphs $G = (V, E, \boldsymbol{u})$ and $\overline{G} = (V, \overline{E}, \overline{\boldsymbol{u}})$, by an* embedding *$f$ of G into $\overline{G}$ we mean a multicommodity flow $f = (f_{e_1}, \ldots, f_{e_{|E|}})$ in $\overline{G}$ with $|E|$ commodities indexed by the edges of G, such that for each $1 \leq i \leq |E|$, the flow f_{e_i} routes in $\overline{G}$ u_{e_i} units of flow between the endpoints of e_i.*

One may view an embedding f of G into $\overline{G}$ as a concurrent flow in $\overline{G}$ corresponding to a demand graph given by G, i.e., source-sink pairs and corresponding demands are given by the endpoints of edges of G and their corresponding capacities. Note that the above definition does not require that f is feasible. We proceed to the definition of embedability.

Definition 5.2.2. *For given $t \geq 1$, graphs $G = (V, E, \boldsymbol{u})$, and $\overline{G} = (V, \overline{E}, \overline{\boldsymbol{u}})$, we say that G* is t-embeddable into *$\overline{G}$ if there exists an embedding f of G into $\overline{G}$ such that for all $e \in \overline{E}$, $|f(e)| \leq t\overline{u}_e$. We say that G* is embeddable into *$\overline{G}$ if G is 1-embeddable into $\overline{G}$.*

Intuitively, the fact that G is t-embeddable into $\overline{G}$ means that we can fractionally pack all the edges of G into $\overline{G}$ having all its capacities $\overline{u}$ multiplied by t. Also, it is easy to see that for given graphs G and $\overline{G}$, one can always find the smallest possible t such that G is t-embeddable into $\overline{G}$ by just solving the maximum concurrent flow problem in $\overline{G}$ in which we treat G as a demand graph with the demands given by its capacities.

[3]The fact that we use the notion of demand graph in the definition of both the maximum concurrent flow problem and the generalized sparsest cut problem, is not a coincidence. These two problems turn out to be closely related – see [102] and [19].

5.3 Graph Decompositions and Fast Approximation Algorithms for Cut-based Problems

The central concept of our approach is the notion of an $(\alpha, \mathcal{G})$-decomposition. This notion is a generalization of the cut-based graph decomposition introduced by Räcke [115].

Definition 5.3.1. *For any $\alpha \geq 1$ and some family of graphs $\mathcal{G}$, by an $(\alpha, \mathcal{G})$-*decomposition *of a graph $G = (V, E, \boldsymbol{u})$ we mean a set of pairs $\{(\lambda^i, G^i)\}_i$, where for each i, $\lambda^i > 0$ and $G^i = (V, E^i, \boldsymbol{u}^i)$ is a graph in $\mathcal{G}$, such that:*

(a) $\sum_i \lambda^i = 1$

(b) G is embeddable into each G^i

(c) There exist embeddings f^i of each G^i into G such that for each $e \in E$,

$$\sum_i \lambda^i |f^i(e)| \leq \alpha u_e.$$

*Moreover, we say that such decomposition is k-*sparse *if it consists of at most k different G^is.*

In other words, an $(\alpha, \mathcal{G})$-decomposition is constituted by a convex combination $\{(\lambda^i, G^i)\}_i$ of graphs from the family $\mathcal{G}$ such that one can pack G into each of these G^is, and one can pack the convex combination of these G^is into G while exerting a congestion of at most α in G.

The crucial property of $(\alpha, \mathcal{G})$-decomposition is that it captures α-approximately the cut structure of the graph G. We formalize this statement in the following easy to prove fact.

Fact 5.3.2. *For any $\alpha \geq 1$, graph $G = (V, E, \boldsymbol{u})$, and a family of graphs $\mathcal{G}$, if $\{(\lambda^i, G^i)\}_i$ is an $(\alpha, \mathcal{G})$-decomposition of G then for any cut C of G:*

(lowerbounding) *for all i, the capacity $u^i(C)$ of C in G^i is at least its capacity $u(C)$ in G;*

(upperbounding in expectation) $\mathbb{E}_{\vec{\lambda}}[u(C)] := \sum_i \lambda^i u^i(C) \leq \alpha u(C)$.

Note that the condition (a) from Definition 5.3.1 implies that $\{(\lambda_i, G_i)\}_i$ is a convex combination of the graphs $\{G^i\}_i$. Therefore, one of the implications of the Fact 5.3.2 is that for any cut C in G, not only the capacity of C in every G^i is lowerbounded by its capacity $u(C)$ in G, but also there always exists a graph G^j in which the capacity of C is at most $\alpha u(C)$. As it turns out, this property alone allows one to reduce a task of $\alpha\beta$-approximation of any cut-based minimization problem in G, any $\beta \geq 1$, to a task of β-approximation of this problem in each of G^is.

To see why this is the case, consider an instance P of some cut-based minimization problem and let f_P be the function corresponding to this instance (cf. (5.2)). Let C^i, for each i, be some β-optimal solution to the instance P in G^i – we can obtain these solutions by just running the promised β-approximation algorithm for G^is. Now, let $C := \text{argmin}_i u(C^i) f_P(C^i)$ be the one among solutions C^i that constitutes the best solution for the original G. We claim that C is a $\alpha\beta$-approximate solution for G. This is so, since if C^* is the optimal solution for G, then – as we observed above – there exists j such that $u^j(C^*) \leq \alpha u(C^*)$. But this means that

$$\begin{aligned} u(C)f_P(C) &\leq u(C^j)f_P(C^j) \leq u^j(C^j)f_P(C^j) \\ &\leq \beta u^j(C^*)f_P(C^*) \leq \alpha\beta u(C^*)f_P(C^*) = \alpha\beta OPT, \end{aligned}$$

where the first inequality follows from definition of C, the second one from cut lowerbounding property of $(\alpha, \mathcal{G})$-decomposition, and the third one from the fact that C^j was a β-approximate solution for G^j. So, C is indeed an $\alpha\beta$-approximate solution for G.

In the above, we reduce the task of approximation of a cut-based minimization problem to a task of approximation of this problem in all G^is. However, it turns out that if one is willing to settle for Monte Carlo-type approximation guarantees, one just need to approximately solve this problem in only a *small number* of G^is. To make this statement precise, let us introduce the following definition.

Definition 5.3.3. *For a given $G = (V, E, u)$, $\alpha \geq 1$, and $1 \geq p > 0$, we say that a collection $\{G^i\}_i$ of random graphs[4] $G^i = (V, E^i, u^i)$, α-*preserves the cuts of G with probability p *if for every cut C of G:*

(lowerbounding) *for all i, the capacity $u^i(C)$ of C in G_i is at least its capacity $u(C)$ in G;*

(probabilistic upperbounding) *with probability at least p, there exists i such that $u^i(C) \leq \alpha u(C)$.*

With a slight abuse of notation, we will say that a random graph G' is α-preserving the cuts of G with probability p if the corresponding singleton family $\{G'\}$ is doing it. Now, a useful connection between $(\alpha, \mathcal{G})$-decomposition of graph G and obtaining graphs $O(\alpha)$-preserving the cuts of G with some probability is given by the following fact whose proof is a straight-forward application of Markov's inequality.

Fact 5.3.4. *Let $G = (V, E, \boldsymbol{u})$, $\alpha \geq 1$, $1 > p > 0$ and let $\{(\lambda^i, G^i)\}_i$ be some $(\alpha, \mathcal{G})$-decomposition of G. If G' is a random graph chosen from the set $\{G^i\}_i$ according to distribution given by λ^is i.e. $Pr[G' = G^i] = \lambda^i$, then G' 2α-preserves cuts of G with probability $1/2$.*

Therefore – as we will formally prove later – for any $\beta \geq 1$, we can obtain a Monte Carlo $2\alpha\beta$-approximation algorithm for any minimization cut-based problem

[4]We formalize the notion of random graphs by viewing each G^i as a graph on vertex set V chosen according to some underlying distribution over all such graphs.

in G, by β-approximating it in a sample of $O(\ln |V|)$ G_is that were chosen according to distribution described by λ^is.

5.3.1 Finding a Good $(\alpha, \mathcal{G})$-decomposition Efficiently

In the light of the above discussion, we focus our attention on the task of developing a $(\alpha, \mathcal{G})$-decomposition of graphs into some family $\mathcal{G}$ of structurally simple graphs that enjoys relatively small value of α. Of course, since our emphasis is on obtaining algorithms that are very efficient, an important aspect of the decomposition that we will be interested in is the time needed to compute it.

With this goal in mind, we start by considering the following theorem due to Räcke [115] that describes the quality of decomposition one can achieve when decomposing the graph G into a family $\mathcal{G}_T$ of its decomposition trees (cf. [115] for a formal definition of a decomposition tree).

Theorem 5.3.5 ([115]). *For any graph $G = (V, E, \boldsymbol{u})$, an $\widetilde{O}(m)$-sparse $(O(\log n), \mathcal{G}_T)$-decomposition of G can be found in polynomial time, where $n = |V|$ and $m = |E|$.*

Due to structural simplicity of decomposition trees, as well as, the quality of cut preservation being the best – up to a constant – achievable in this context, this theorem was used to design good approximation algorithms for a number of cut-based minimization problems.

Unfortunately, from our point of view, the usefulness of Räcke's decomposition is severely limited by the fact that the time needed to construct it is not acceptable when one is aiming at obtaining fast algorithms. More precisely, the running time of the decomposing algorithm of [115] is dominated by $\widetilde{O}(m)$ executions of the algorithm of Fakcharoenphol, Talwar and Rao [62] that solves the minimum communication cost tree problem with respect to a cost function devised from the graph G (cf. [115] for details). Each such execution requires, in particular, computation of the shortest-path metric in G with respect to some length function, which employs the well-known all-pair shortest path algorithm [66, 137, 7] running in $O(\min\{mn, n^{2.376}\})$ time. This results in – entirely acceptable when one is just interested in obtaining polynomial-time approximation algorithms, but prohibitive in our case – $\widetilde{O}(m \min\{mn, n^{2.376}\})$ total running time.

One could try to obtain a faster implementation of Räcke's decomposition by using a nearly-linear time low-average-stretch spanning tree construction due to Abraham, Bartal, and Neiman [1] (cf. Theorem 2.6.3) in place of the algorithm of [62]. This would lead to an $\widetilde{O}(m^2)$ time decomposition of G into its spanning trees that has a – slightly worse – quality of $\widetilde{O}(\log n)$ instead of the previous $O(\log n)$. However, although we will use [1] in our construction (cf. Section 5.5.2), $\widetilde{O}(m^2)$ time is still not sufficiently fast for our purposes.

The key idea for circumventing this running time bottleneck is allowing ourselves more flexibility in the choice of the family of graphs into which we will decompose G. Namely, we will be considering $(\alpha, \mathcal{G})$-decompositions of G into objects that are still structurally simpler than G, but not as simple as trees.

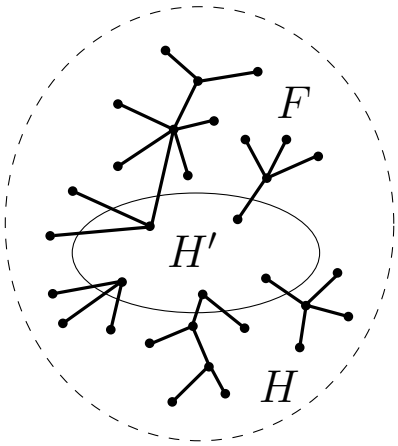


Figure 5-2: An example of a j-tree H, its core H', and its envelope F consisting of bold edges.

To this end, for $j \geq 1$, we say that a graph $H = (V_H, E_H, u_H)$ is a *j-tree* (cf. Figure 5-2) if it is a connected graph being a union of: a subgraph H' of H induced by some vertex set $V'_H \subseteq V_H$ with $|V'_H| \leq j$; and of a forest F on V_H whose each connected component has exactly one vertex in V'_H. We will call the subgraph H' the *core* of H and the forest F the *envelope* of H.[5]

Now, if we define $\mathcal{G}_V[j]$ to be the family of all the j-trees on the vertex set V, the following theorem – being the heart of our framework – holds. Its proof appears in section 5.5.

Theorem 5.3.6. *For any graph $G = (V, E, \boldsymbol{u})$ and $t \geq 1$, we can find in time $\widetilde{O}(tm)$ a t-sparse $(\widetilde{O}(\log n), \mathcal{G}_V[\widetilde{O}(\frac{m \log U}{t})])$-decomposition $\{(\lambda_i, G_i)\}_i$ of G, where $m = |E|$, $n = |V|$, and U is the capacity ratio of G. Moreover, the capacity ratio of each G_i is $O(mU)$.*

Intuitively, the above theorem shows that if we allow $j = \widetilde{O}(\frac{m \log U}{t})$ to grow, the sparsity of the corresponding decomposition of G into j-trees – and thus the time needed to compute it – will decrease proportionally.[6]

Note that a 1-tree is just an ordinary tree, so by taking t in the above theorem sufficiently large, we obtain a decomposition of G into trees in time $\widetilde{O}(m^2 \log U)$. Therefore, we see that compared to the decomposition result of Räcke (cf. Theorem 5.3.5), we obtain in this case a decomposition of G with more efficient implementation and into objects that are even simpler than decomposition trees, but at a cost of slightly worse quality.

[5]Note that given a j-tree we can find its envelope in linear time. Therefore, we will always assume that the j-trees we are dealing with have their envelopes and cores explicitly marked.

[6]Interestingly, such a trade-off is somewhat reminiscent of the trade-off achieved by Spielman and Teng [129, 130] between the number of additional edges of an ultrasparsifier and the quality of spectral approximation provided by it – see Section 2.6.

However, the aspect of this theorem that we will find most useful is the above-mentioned trade-off between the simplicity of the j-trees into which the decomposition decomposes G and the time needed to compute it. This flexibility in the choice of t can be utilized in various ways.

For instance, one should note that, in some sense, the core of a j-tree H captures all the non-trivial cut-structure of H. In particular, it is easy to see that maximum flow computations in H essentially reduce to maximum flow computations in H's core. So, one might hope that for some cut-based problems the complexity of solving them in H is proportional to the complexity of solving them in the – possibly much smaller than H – core of H (this is, for example, indeed the case for the balanced separator and the sparsest cut problems – see section 5.4.3). Thus one could use Theorem 5.3.6 – for appropriate choice of t – to get a faster algorithm for this kind of problems by just computing first the corresponding decomposition and then leveraging the existing algorithms to solve the given problem on a small number of sampled j-trees, while paying an additional $\widetilde{O}(\log n)$ factor in the approximation quality for this speed-up.

Even more importantly, our ability to choose in Theorem 5.3.6 sufficiently small value of t, as well as, the cut sparsification technique (cf. Theorem 3.5.1) and recursive application of the theorem to the cores of $\widetilde{O}(\frac{m \log U}{t})$-trees sampled from the computed decomposition, allows us to establish the following theorem – its proof appears in section 5.6.

Theorem 5.3.7. *For any $1 \geq l \geq 0$, integral $k \geq 1$, and any graph $G = (V, E, \boldsymbol{u})$, we can find in $\widetilde{O}(m + 2^k n^{(1+\frac{1-l}{2^k-1})} \log U)$ time a collection of $(2^{k+1} \ln n)$ n^l-trees $\{G^i\}_i$ that $(\log^{(1+o(1))k} n)$-preserve the cuts of G with high probability. Moreover, the capacity ratio of each G^i is $n^{(2+o(1))k}U$, where U is the capacity ratio of G, $n = |V|$, and $m = |E|$.*

As one can see, the above theorem allows obtaining a collection of j-trees that α-preserve cuts of G – for *arbitrary* $j = n^l$ – in time *arbitrarily* close to nearly-linear, but at a price of α growing accordingly as these two parameters decrease.

Note that one can get a cut-preserving collection satisfying the requirements of the theorem by just finding an $(\widetilde{O}(\log n), n^l)$-decomposition of G via Theorem 5.3.6 and sampling – in the spirit of Fact 5.3.4 – $O(\log n)$ n^l-trees from it so as to ensure that each cut is preserved with high probability. Unfortunately, the running time of such procedure would be too large. Therefore, our approach to establishing Theorem 5.3.7 can be heuristically viewed as an algorithm that in case when the time required by Theorem 5.3.6 to compute a decomposition of G into n^l-trees is not acceptable, does not try to compute and sample from such decomposition directly. Instead, it performs its sampling by finding a decomposition of G into j-trees, for some value of j bigger than n^l, then samples a j-tree from it and recurses on this sample. Now, the desired collection of n^l-trees is obtained by repeating this sampling procedure enough times to make sure that the cut preserving probability is high enough, with the value of j chosen so as to bound the number of recursive calls by $k-1$. Since each recursive call introduces an additional $\widetilde{O}(\log n)$ distortion in the faithfulness of the reflection of the cut structure of G, the quality of the cut preservation of the collection generated via such algorithm will be bounded by $\log^{(1+o(1))k} n$.

5.4 Applications

We proceed to demonstrating how the tools and ideas presented in the previous section lead to a fast approximation algorithms for cut-based graph problems. In particular, we apply our techniques to the (generalized) sparsest cut and the balanced separator problems.

The following theorem encapsulates our way of employing the framework.

Theorem 5.4.1. *For any $\alpha \geq 1$, integral $k \geq 1$, $1 \geq l \geq 0$, and undirected graph $G = (V, E, \boldsymbol{u})$ with $n = |V|$, $m = |E|$, and U being its capacity ratio, we can find in $\widetilde{O}(m + 2^k n^{(1+\frac{1-l}{2^k-1})} \log U)$ time, a collection of $2^{k+1} \ln n$ n^l-trees $\{G^i\}_i$, such that for* any *instance P of* any *cut-based minimization problem $\mathcal{P}$ the following holds with high probability. If $\{C^i\}_i$ is a collection of cuts of G with each C^i being some α-optimal solution to P on the n^l-tree G^i, then at least one of C^i is an $(\alpha \log^{(1+o(1))k} n)$-optimal solution to P on the graph G.*

When we take l equal to zero, the above theorem implies that if we have an α-approximation algorithm for a given cut-based problem that works only on tree instances and runs in $T(m, n, U)$ time then, for any $k \geq 1$, we can get an $(\alpha \log^{(1+o(1))k} n)$-approximation algorithm for it in general graphs and the running time of this algorithm will be just $\widetilde{O}(m + k2^k n^{(1+1/(2^k-1)} \log U) + (2^{k+1} \ln n) T(m, n, U)$. Note that the computational overhead introduced by our framework, as k grows, quickly approaches nearly-linear. Therefore, if we are interested in designing fast poly-logaritmic approximation algorithms for some cut-based minimization problem, we can just focus our attention on finding a fast approximation algorithm for its tree instances.

Also, an interesting feature of our theorem is that the procedure producing the graphs $\{G^i\}_i$ is completely oblivious to the cut-based problem we want to solve – the fact that this problem is a cut-based minimization problem is all we need to make our approach work.

Proof of Theorem 5.4.1. We produce the desired collection $\{G^i\}_i$ by just employing Theorem 5.3.7 with the desired value of $k \geq 1$ to find in time $\widetilde{O}(m+2^k n^{(1+\frac{1-l}{2^k-1})} \log U)$ a set of $t = (2^{k+1} \ln n)$ n^l-trees that $(\log^{(1+o(1))k} n)$-preserve the cuts of G with high probability and output them as $\{G^i\}_i$.

Now, to prove the theorem, let us define C to be the cut that minimizes the quantity $u(C^i) f_P(C^i)$ among all the t solutions $C^1, \ldots, C^t$ found, where f_P is the function corresponding to the instance P of a problem $\mathcal{P}$ we are solving (cf. equation (5.2)). Clearly, by definition of $(\log^{(1+o(1))k} n)$-preservation of the cuts, the capacity of C – and thus the quality of the solution corresponding to it – can only improve in G i.e.

$$u(C) f_P(C) \leq u^j(C) f_P(C),$$

where G^j is the n^l-tree to which C corresponds and, for any i, $\boldsymbol{u}^i$ denotes the capacity vector of G^i.

Moreover, if we look at an optimum solution C^* to our problem in G – i.e. $C^* = \arg\min_{\emptyset \neq C \subset V} u(C) f_P(C)$ – then, with high probability, in at least one of the G^is,

say in $G^{j'}$, C^* has the capacity at most $(\log^{(1+o(1))k} n)$ times larger than its original capacity $u(C^*)$ in G. As a result, for the α-optimal solution $C^{j'}$ found by the algorithm in $G^{j'}$ it will be the case that

$$u(C^{j'})f_P(C^{j'}) \leq u^{j'}(C^{j'})f_P(C^{j'}) \leq \alpha u^{j'}(C^*)f_P(C^*) \leq \alpha(\log^{(1+o(1))k} n)u(C^*)f_P(C^*),$$

where the first inequality follows from the fact that $\{G^i\}_i$ $(\log^{(1+o(1))k} n)$-preserve cuts of G.

But, by definition of C, we have

$$u(C)f_P(C) \leq u(C^{j'})f_P(C^{j'}).$$

So, by noting that $u(C^*)f_P(C^*)$ is the objective value of an optimal solution to our instance, we get that C is indeed an $(\alpha \log^{(1+o(1))k} n)$-optimal solution to P with high probability. □

5.4.1 Computing Maximum Concurrent Flow Rate on Trees

As Theorem 5.4.1 suggests, we should focus on designing fast approximation algorithms for tree instances of our problems. The basic tool we will use in this task is the ability to compute maximum concurrent flow rate on trees in nearly-linear time. The main reason for existence of such a fast algorithm stems from the fact that if we have a demand graph $D = (V, E_D, \boldsymbol{d})$ and a tree $T = (V, E_T, \boldsymbol{u}^T)$, there is a unique way of satisfying these demands in T. Namely, for each demand edge $e \in E_D$ the flow of d_e units of corresponding commodity has to be routed along the unique path $\mathsf{path}_T(e)$ joining two endpoints of e in the tree T. As a result, we know that if we want to route in T a concurrent flow of rate $\theta = 1$ then the total amount of flow flowing through a particular edge h of T is equal to

$$u^T[D](h) := \sum_{e \in E_D, h \in \mathsf{path}_T(e)} d_e.$$

Interestingly, we can compute $u^T[D](h)$ for all $h \in E_T$ in nearly-linear time.

Lemma 5.4.2. *For any tree $T = (V, E_T, \boldsymbol{u}^T)$ and demand graph $D = (V, E_D, \boldsymbol{d})$, we can compute $u^T[D](h)$, for all $h \in E_T$, in $\widetilde{O}(|E_D| + |V|)$ time.*

Proof. To compute all the values $u^T[D](h)$, we adapt the approach to computing the stretch of edges outlined by Spielman and Teng in [130].

For a given tree $T' = (V', E_{T'}, \boldsymbol{u}^{T'})$ and a demand graph $D' = (V', E_{D'}, \boldsymbol{d}')$ corresponding to it, let us define the *size* $S_{D'}(T')$ of T' (with respect to D') as $S_{D'}(T') := \sum_{v \in V'}(1 + d_{D'}(v))$, where $d_{D'}(v)$ is the degree of the vertex v in D'. Let us define a *splitter* $v^*_{D'}(T')$ of T' (with respect to D') to be a vertex of T' such that each of the trees $T'_1, \ldots, T'_q$ obtained from T' by removal of $v^*_{D'}(T')$ and all the

adjacent edges has its size $S_{D'}(T'_i)$ being at most one half of the size of T' i.e.

$$S_{D'}(T'_i) \leq S_{D'}(T')/2 = |E_{D'}| + |V'|/2,$$

for each i. It is easy to see that such a splitter $v^*(T')$ can be computed in a greedy fashion in $O(|V'|) = O(S_{D'}(T'))$ time.

Our algorithm for computing $u^T[D]$s works as follows. It starts with finding a splitter $v^* = v^*_D(T)$ of T. Now, let E_0 be the set of demand edges $e \in E_D$ such that $\mathsf{path}_T(e)$ contains v^*. Also, for $1 \leq i \leq q$, let $E_i \subseteq E_D$ be the set of edges $e \in D$ for which $\mathsf{path}_T(e)$ is contained entirely within T_i. Note that this partition of edges can be easily computed in $O(|E_D| + |V|) = O(S_{D'}(T'))$ time. Also, let us define D_i to be the demand graph being the subgraph of D spanned by the demand edges from E_i (with demands inherited from D).

As a next step, our algorithm computes in a simple bottom-up fashion $u^T[D_0](h)$ for each $h \in E_T$. Then, for $1 \leq i \leq q$, it computes recursively the capacities $u^{T_i}[D_i]$ of all the edges of T_i – note that, by definition of D_i, $u^{T_i}[D_i](h) = u^T[D_i](h)$ for any edge h of T_i. Finally, for each $h \in E_T$, we output $u_T[D](h) = \sum_i u^T[D_i](h)$. Note that, since for $i \geq 1$, $u^T[D_i](h) \neq 0$ only if h is a part of T_i, the sum $\sum_i u^T[D_i](h)$ has at most two non-zero components and thus all the values $u^T[D](h)$ can be computed in $O(|E_D| + |V|) = O(S_D(T))$ time. Furthermore, the fact that, by our choice of v^*, $S_{D_i}(T_i) \leq S_D(T)/2$ implies that the depth of recursion is at most $\log S_D(T)$ and the whole algorithm runs in $\widetilde{O}(S_D(T)) = \widetilde{O}(|E_D| + |V|)$ time, as desired. □

Now, the crucial observation to be made is that the best achievable flow rate θ^* of the maximum concurrent flow corresponding to the demand graph $D = (V, E_D, \boldsymbol{d})$ is equal to $\min_{h \in E_T} \frac{u^T_h}{u^T[D](h)}$. Therefore, Lemma 5.4.2 implies the following corollary.

Corollary 5.4.3. *For any tree $T = (V, E_T, \boldsymbol{u}^T)$ and demand graph $D = (V, E_D, \boldsymbol{d})$, we can find in $\widetilde{O}(|E_D| + |V|)$ time the optimal maximum concurrent flow rate θ^* and an edge $h^* \in E_T$ such that $\theta^* = \frac{u^T_{h^*}}{u^T[D](h^*)}$.*

Note that we only compute the optimal flow rate of the concurrent flow and not the actual flows. In some sense, this is unavoidable – one can easily construct an example of maximum concurrent flow problem on tree, where the representation of any (even only approximately) optimal flow has size $\Omega(|E_D||V|)$ – see Figure 4-2 in Chapter 4.

5.4.2 Generalized Sparsest Cut Problem

We proceed to designing a fast approximation algorithm for the generalized sparsest cut problem. We start by noticing that for a given tree $T = (V, E_T, \boldsymbol{u}^T)$, demand graph $D = (V, E_D, \boldsymbol{d})$, and an edge h of this tree, the quantity $\frac{u^T_h}{u^T[D](h)}$ is exactly the (generalized) sparsity of the cut that cuts in T only the edge h. Therefore, Corollary 5.4.3 together with the fact that the sparsity of the sparsest cut is always an upper bound on the maximum concurrent flow rate achievable (cf. [102, 19]), gives us the following corollary.

Corollary 5.4.4. *For any given tree $T = (V, E, \boldsymbol{u})$ and demand graph $D = (V, E_D, \boldsymbol{d})$, an optimal solution to the generalized sparsest cut problem can be computed in $\widetilde{O}(|E_D| + |V|)$ time.*

Now, by applying Theorem 5.4.1 together with a preprocessing step of cut sparsification of D (cf. Corollary 2.6.2), we are able to obtain a poly-logarithmic approximation for the generalized sparsest cut problem in close to linear time.

Theorem 5.4.5. *For any graph $G = (V, E, \boldsymbol{u})$, demand graph $D = (V, E_D, \boldsymbol{d})$, and integral $k \geq 1$, there exists a Monte Carlo $\log^{(1+o(1))k} n$-approximation algorithm for generalized sparsest cut problem that runs in time $\widetilde{O}(m + |E_D| + 2^k n^{(1+1/(2^k-1))} \log U)$, where $n = |V|$, $m = |E|$, and U is the capacity ratio of G.*

Proof. We start by employing Corollary 2.6.2 with δ equal to 1, to sparsify both G – to obtain a graph $\widetilde{G} = (V, \widetilde{E}, \widetilde{\boldsymbol{u}})$) – and the demand graph D – to obtain a demand graph $\widetilde{D} = (V, \widetilde{E}_D, \widetilde{\boldsymbol{d}})$ – in total time of $\widetilde{O}(m + |E_D|)$. Note that computing sparsity of a cut with respect to these sparsified versions of G and D leads to a 4-approximate estimate of the real sparsity of that cut. Since this constant-factor approximation is acceptable for our purposes, we can focus on approximating the sparsest cut problem with respect to $\widetilde{G}$ and $\widetilde{D}$. To this end, we just use Theorem 5.4.1 together with Corollary 5.4.4 and, since both $|\widetilde{E}|$ and $|\widetilde{E}_D|$ have $\widetilde{O}(n)$ edges, our theorem follows. □

5.4.3 Balanced Separator and Sparsest Cut Problem

We turn our attention to the balanced separator problem. Analogously to the case of the generalized sparsest cut problem above, to employ our approach we need an efficient algorithm for the tree instances of the balanced separator problem. Unfortunately, although we can solve this problem on trees optimally via dynamic programming, there seems to be no algorithm that does it very efficiently – ideally, in nearly-linear time. Therefore, we circumvent this problem by settling for a fast but approximate solution.

Namely, we use the result of Sherman [123] who shows that, for any $\varepsilon > 0$, the balanced separator problem – as well as the sparsest cut problem – can be $O(\sqrt{\log n/\varepsilon})$-approximated in a graph G – with n vertices and m edges – in time $\widetilde{O}(m + n^{4/3+\varepsilon})$. This running time corresponds to sparsifying G and then using perform (approximate) maximum flow computations on a sequence of n^{ε} graphs that are derived in a certain way from the sparsified version of G. As these computations can be performed using the maximum flow algorithm from Chapter 3, the running time $\widetilde{O}(m + n^{4/3+\varepsilon})$ follows.

Unfortunately, $\widetilde{O}(m + n^{4/3+\varepsilon})$ running time is still not sufficiently fast for our purposes. At this point, however, we recall that maximum flow computation in a j-tree reduces to the task of finding the maximum flow in its core. So, if we want to perform a maximum flow computation on a j-tree that has its core sparsified (i.e. the core has only $\widetilde{O}(j)$ edges), the real complexity of this task is proportional to j as opposed to being proportional to n. This motivates us to obtaining an implementation

of Sherman's algorithm on j-trees that achieves better running time. This is captured in the following lemma.

Lemma 5.4.6. *For any j-tree $G = (V, E, \boldsymbol{u})$ and $\varepsilon > 0$, one can $O(\sqrt{\log n/\varepsilon})$-approximate the balanced separator and the sparsest cut problems in $\widetilde{O}(m + n^{\varepsilon}(n + j^{4/3}))$ time, where $m = |E|$ and $n = |V|$.*

Proof. We start by using Corollary 2.6.2 with $\delta = 1$ to sparsify the core of our j-tree G. This ensures that the resulting j-tree $\widetilde{G}$ has a core with $\widetilde{O}(j)$ edges. Also, it is easy to see that we can focus on approximating our problems in this graph since any approximation obtained for $\widetilde{G}$ leads to an approximation for G that is only by at most a constant factor worse.

For given $\varepsilon > 0$, the algorithm of Sherman reduces the task of $O(\sqrt{\log n/\varepsilon})$-approximation of the balanced partition and the sparsest cut problems in the graph $\widetilde{G}$ to solving a sequence of n^{ε} instances of the following problem[7]. We are given a graph $\widehat{G}$ being $\widetilde{G}$ with a source s and sink t added to it. We also add *auxiliary* edges that connect these two vertices with the original vertices corresponding to $\widetilde{G}$. The capacities of the auxiliary edges can be arbitrary – in particular, they can be zero which means that the corresponding edge is not present – but we require that the value of the s-t cut $C = \{s\}$ is at most $n/2$ (thus the throughput of the maximum s-t flow is also bounded by this value). The task is to find a constant approximation to the maximum s-t flow in the graph $\widehat{G}$.

Our goal is to show that we can solve any instance of the above problem in $\widetilde{O}(n + j^{4/3})$ time – this will imply our desired $\widetilde{O}(m + n^{\varepsilon}(n + j^{4/3}))$ total running time of Sherman's $O(\sqrt{\log n/\varepsilon})$-approximation algorithm and thus yield the proof of the lemma.

To this end, we design a method of fast compression of the graph $\widehat{G}$ to make it only consist of the core of $\widetilde{G}$, the source s and the sink t, and the corresponding auxiliary edges (with modified capacities). This method will allow efficient – i.e. linear time – recovering from any flow in the compressed graph a flow in the original graph $\widehat{G}$, such that if the flow in the compressed graph was an approximately maximum flow then so is the recovered one (with the same approximation factor). So, by running the maximum flow algorithm from Chapter 3 on this compressed graph and recovering the flow in $\widehat{G}$ from the computed flow, we will obtain an approximately maximum flow in $\widehat{G}$ in time $\widetilde{O}(n + j^{4/3})$, as desired.

Our compression procedure works in steps – each such step is a local transformation that reduces the number of vertices and edges of the graph by at least one and can be implemented in constant time. In each transformation we consider a vertex v in the current version $\widehat{G}'$ of $\widehat{G}$ such that v has exactly one non-auxiliary edge e incident to it and this edge is a part of an envelope of $\widetilde{G}$. As we will see, $\widehat{G}'$ will be always a subgraph of the graph $\widehat{G}$ and thus one can convince oneself that – as long as $\widehat{G}'$ still contains some edges of the envelope of $\widetilde{G}$ – we can always find a vertex v as above.

[7]For our purposes, we use a slight generalization of the problem that is actually considered in [123].

Assume first that there is at most one auxiliary edge that is incident to v. (For the sake of the argument, let us use here a convention that this single auxiliary edge e' is always present, but might have a capacity of zero.) In this case, we just contract the edge e and set the new capacity of e' to be the minimum of its previous capacity and the capacity of e. We claim now that given any flow f'' in the graph $\widehat{G}''$ obtained through this contraction of e, we can extend it in constant time to a flow f' in the graph $\widehat{G}'$ of the same value. To achieve this, we just transplant the flow of $|f''(e')|$ units that f'' routes over the edge e' in $\widehat{G}''$, to a flow in $\widehat{G}'$ routed through the edges e and e'. Note that by definition of the capacity of e' in $\widehat{G}''$, the transplanted flow f' is feasible in $\widehat{G}'$ and has the same value as f'' had in $\widehat{G}''$. Moreover, one can see that the value of the maximum flow in $\widehat{G}'$ can be at most the value of the maximum flow in $\widehat{G}''$ since any minimum s-t cut in $\widehat{G}''$ can be easily extended to a minimum s-t cut in $\widehat{G}'$ that has the same capacity. So, if f'' was approximately maximum flow in $\widehat{G}''$ then so is f' in $\widehat{G}'$.

Now, we deal with the case when there are two auxiliary edges e',e'' incident to v (wlog, let us assume that the capacity u'' of e'' is not bigger than the capacity u' of e'). To this end, we note that if we route u'' units of flow from s to t along the path consisting of the edges e',e'' then there still exists a maximum flow in $\widehat{G}'$ such that its flow-path decomposition contains the flow-path corresponding to our pushing of u'' units of flow over the edges e', e''. This implies that if we reduce the capacity of e' and e'' by u'' – thus reducing the capacity of e'' to zero and removing it – and find the approximate maximum flow f'' in the resulting graph, we can still recover an approximately maximum flow in $\widehat{G}'$ by just adding the above-mentioned flow-path to f''. But, since in the resulting graph v has at most one auxiliary edge incident to it – namely, e' – we can use our transformation from the previous case to compress this graph further. This will reduce the number of vertices and edges of $\widehat{G}'$ by at least one while still being able to recover from an approximately maximum flow in the compressed graph $\widehat{G}''$ an approximately maximum flow in $\widehat{G}'$.

By noting that the compression procedure stops only when $\widehat{G}'$ contains no more edges of the envelope of $\widetilde{G}$ and thus the final compressed graph indeed consists of only (sparsified) core of $\widetilde{G}$, source s, and t, the lemma follows.

□

Before we proceed further, we note that we can compute very quickly a very rough, but polynomial in $|V|$, approximation to the graph partitioning problems of our interest.

Lemma 5.4.7. *For any graph $G = (V, E, \boldsymbol{u})$, we can find a $|V|^2$-approximation to the sparsest cut and the balanced separator problems in time $\widetilde{O}(|E|)$.*

Proof. Consider the case when we are interested in the sparsest cut problem (we will present variation for the balanced separator problem shortly). We sort the edges of G non-increasingly according to their capacities. Let $e_1, \ldots, e_{|E|}$ be the resulting ordering. Let r^* be the smallest r such that the set $\{e_1, \ldots, e_r\}$ contains a spanning tree T of G. It is easy to see that we can find such r^* in $\widetilde{O}(|E|)$ time using the union-find data structure.

Now, let us look at the cut C in G that cuts only the edge e_{r^*} in T. Since no vertex of G can have degree bigger than $|V|$, the sparsity of C is at most $u_{e_{r^*}}|V|$. On the other hand, any cut in G has to cut at least one edge e_r with $r \leq r^*$. Therefore, we know that the sparsity of the optimal cut in G has to be at least $u_{e_{r^*}}/|V|$. This implies that C is the desired $|V|^2$-approximation of the sparsest cut of G.

In case of balanced separator problem, we define r^* to be the smallest r such that the largest connected component of the subgraph E_r spanned by the edges $\{e_1, \ldots, e_r\}$ has its size bigger than $(1-c)|V|$, where c is the desired balance constant of our instance of the problem. Once again, one can easily find such r^* in $\widetilde{O}(|E|)$ time by employing union-find data structure. Let F be the connected component of E_{r^*} containing e_{r^*}. Note that by minimality of r^*, removing e_{r^*} from F disconnects it into two connected pieces and at least one of these pieces, say F', has to have at least $(1-c)|V|/2$ and at most $(1-c)|V|$ vertices. Let C be the cut corresponding to F'. Clearly, the sparsity of C is at most $u_{e_{r^*}}|V|$. Furthermore, any cut C^* with $\min\{|C^*|, |\overline{C^*}\} \geq c|V|$ has to cut some edge e_r with $r \geq r^*$. Therefore, the optimal solution can have sparsity at most $u(e_{r^*})/|V|$. This implies that C is a c'-balanced separator with $c' = \min\{(1-c)/2, c\}$ and thus constitutes our desired $|V|^2$-(pseudo-)approximation for our instance of the balanced separator problem. □

Now, we can use Theorem 5.4.1 and Lemma 5.4.6 – for the right choice of $j = n^l$ – together with a simple preprocessing making the capacity ratio of the graphs we are dealing with polynomially bounded, to obtain the following result.

Theorem 5.4.8. *For any $\varepsilon > 0$, integral $k \geq 1$, and graph $G = (V, E, \boldsymbol{u})$, we can $(\log^{(1+o(1))(k+1/2)} n/\sqrt{\varepsilon})$-approximate the sparsest cut and the balanced separator problems in time $\widetilde{O}(m + 2^k n^{1+\frac{1}{4\cdot 2^k-1}+\varepsilon})$, where $m = |E|$ and $n = |V|$.*

Proof. Let us assume first that the capacity ratio of G is polynomially bounded i.e. it is $n^{O(1)}$. In this case, we just use Theorem 5.4.1 on G with $l = \frac{3\cdot 2^k}{4\cdot 2^k-1}$ to obtain a collection of $(2^{k+1} \ln n)$ n^l-trees $\{G^i\}_i$ in time

$$\widetilde{O}(m + 2^k n^{(1+\frac{1-l}{2^k-1})}) = \widetilde{O}(m + 2^k n^{(1+\frac{1}{4\cdot 2^k-1})}).$$

Next, we compute – using the algorithm from Lemma 5.4.6 – $O(\sqrt{\log n/\varepsilon})$-approximately optimal solutions to our desired problem – being either the sparsest cut or the balanced separator problem – on each of G^is in total time of

$$\widetilde{O}(m + (2^{k+1} \ln n) n^{\varepsilon}(n + n^{4l/3})) = \widetilde{O}(m + 2^k n^{1+\frac{1}{4\cdot 2^k-1}+\varepsilon}).$$

By Theorem 5.4.1 we know that choosing the best one among these solutions will give, with high probability, a $(\log^{(1+o(1))(k+1/2)} n/\sqrt{\varepsilon})$-approximately optimal solution that we are seeking.

To ensure that G has its capacity ratio always polynomially bounded, we devise the following preprocessing procedure. First we use Lemma 5.4.7 to find a cut C being n^2-approximation to the optimal solution of our desired problem. Let ζ be the sparsity of C. We remove all the edges in G that have capacity smaller than ζ/mn^2

and, for all the edge with capacity bigger than ζn, we trim their capacity to ζn. Clearly, the resulting graph G' has its capacity ratio polynomially bounded. Also, the capacity of any cut in G can only decrease in G'.

Now, we just run the approximation algorithm described above on G' instead of G. Also, in the case of approximating the balanced separator problem, we set as our balance constant the value $c'' := \min\{c, c'\}$, where $c' := \min\{|C|, |\overline{C}|\}/n$ and c is the balance constant of our input instance. Since the capacity ratio of G' is polynomially bounded, the running time of this algorithm will be as desired. Moreover, we can assume that the α-approximately optimal cut C' output will have sparsity at most ζ in G' - otherwise, we can just output C as our solution. This means that C' does not cut any edges with trimmed capacity and thus the capacity of C' in G (after putting the removed edges back) can be only by an *additive* factor of ζ/n^2 larger than in G'. But since ζ/n^2 is a lower bound on the sparsity of the optimal solution, we see that C' is our desired $(\log^{(1+o(1))(k+1/2)} n/\sqrt{\varepsilon})$-approximately optimal solution. □

Recall that our main motivation to establishing Lemma 5.4.6 was our inability to solve the balanced separator problem on trees efficiently. However, the obtained trade-off between the running time of the resulting algorithm and the quality of the approximation provided for both the balanced separator and the sparsest cut problems is much better than the one we got for the generalized sparsest cut problem (cf. Theorem 5.4.5). This shows us that sometimes it is beneficial to take advantage of the flexibility in the choice of l given by Theorem 5.4.1 by combining it with an existing fast approximation algorithm for our cut-based problem that allows us to obtain a faster implementation on j-tree instances.

5.5 Proof of Theorem 5.3.6

Let us fix throughout this section the parameter $t \geq 1$, the graph $G = (E, V, \boldsymbol{u})$ to be $(\widetilde{O}(\log n), \mathcal{G}_V[\widetilde{O}(\frac{m \log U}{t})])$-decomposed, $m = |E|$, $n = |V|$, and U equal to the capacity ratio of G. Our proof of Theorem 5.3.6 consists of two main steps. First, we show how to quickly decompose G into a t-sparse $(\widetilde{O}(\log n), \mathcal{H}[\widetilde{O}(\frac{m \log U}{t})])$-decomposition $\{(\lambda^i, H^i)\}_i$, where $\mathcal{H}[j]$ is a family of graphs that we will define shortly. Then, as a second step, we prove that for any graph $H \in \mathcal{H}[j]$ we can efficiently find a graph $\overline{G} \in \mathcal{G}_V[O(j)]$ (i.e. a graph $\overline{G}$ being a $O(j)$-tree) such that H is embeddable into $\overline{G}$ and $\overline{G}$ is 9-embeddable into H. This will imply that both these graphs are equivalent (up to a constant) with respect to their cut-flow structure. As a result, if we consider a decomposition of G into convex combination $\{(\lambda^i, \overline{G}^i)\}_i$, where each graph $\overline{G}^i$ is the graph from $\mathcal{G}_V[O(j)]$ equivalent – in the above sense – to the graph $H^i \in \mathcal{H}[j]$, then we will be able to show that this constitutes the t-sparse $(\widetilde{O}(\log n), \mathcal{G}_V[\widetilde{O}(\frac{m \log U}{t})])$-decomposition of G that we are seeking.

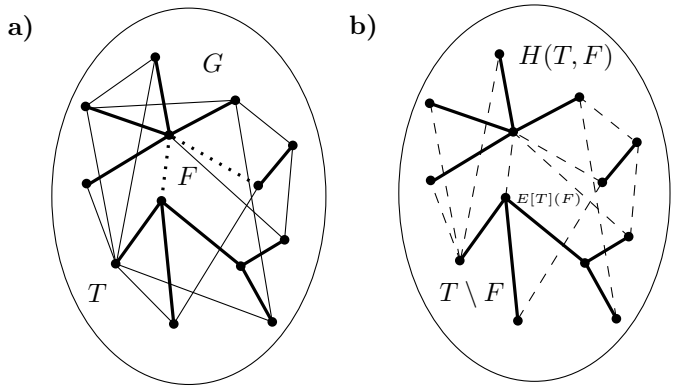


Figure 5-3: **a)** An example of a graph G (solid edges), its spanning tree T (bold edges), and a subset F of edges of T (dotted edges). **b)** Graph $H(T,F)$ corresponding to the example from **a).** The edges of $T \setminus F$ are bold and the edges of the set $E[T](F)$ are dashed.

5.5.1 Graphs $H(T,F)$ and the Family $\mathcal{H}[j]$

To define the family $\mathcal{H}[j]$, consider some spanning tree $T = (V, E_T)$ of G. There is a unique way of embedding G into T. Namely, for each edge $e = (u,v)$ of G we route the corresponding u_e units of flow along the unique u-v path $\mathsf{path}_T(e)$ in T. This implies that if we want G to be embeddable into T then each edge e of the tree T has to have capacity of at least $u^T(e)$, where we define

$$u^T(e) := \sum_{e' \in E : e \in \mathsf{path}_T(e')} u_{e'}.$$

Now, for a subset of edges $F \subseteq E_T$, let us define

$$E[T](F) := \{e \in E \ : \mathsf{path}_T(e) \cap F \neq \emptyset\}.$$

Finally, let $H(T,F) = (V, \overline{E}, \overline{\boldsymbol{u}})$ be a graph (cf. Figure 5-3) with edge set $\overline{E} := E_T \cup E[T](F)$ and the capacities $\overline{u}_e$, for $e \in \overline{E}$, being equal to:

$$\overline{u}_e := \begin{cases} u_e & \text{if } e \in E[T](F) \\ u^T(e) & \text{otherwise.} \end{cases}$$

In other words, $H(T,F)$ is a graph obtained by taking the forest corresponding to the tree T with edges of F removed and adding to it all the edges that are in the set $E[T](F)$ containing the edges of E whose endpoints are in different components of this forest (note that $F \subseteq E[T](F)$). The capacity $\overline{u}_e$ of an edge e of $H(T,F)$ is equal to the capacity $u^T(e)$ inherited from the tree T – if e is from the forest $T \setminus F$; and it is just the original capacity u_e of e otherwise. It is easy to see that, by construction, G embeds into H.

An observation that will turn out to be useful later is that both the capacity vector

$\boldsymbol{u}^T$ and the graph $H(T,F)$ can be constructed very efficiently. This is captured in the following lemma.

Lemma 5.5.1. *Given a spanning tree $T=(V,E_T)$ of G and some $F \subseteq E_T$, we can compute $u^T(e)$ for all $e \in E_T$ and construct the graph $H(T,F)$ in $\widetilde{O}(m)$ time.*

Proof. First, we show how to construct the set $E[T](F)$ in $O(m)$. We do this by first associating with each vertex v a label indicating the connected component of the forest $T \setminus F$ to which v belongs. This can be easily done in $O(n)$ time. Next, we start with $E' := \emptyset$ and for each e edge of G we check whether its both endpoints belong to the same connected component. If this is not the case, we add e to E'. Clearly, this procedure takes $O(m)$ time and the resulting set E' is equal to $E[T](F)$. Now, we obtain $H(T,F)$ by just taking the union of the edges from $E[T](F)$ – with capacities given by the capacity vector $\boldsymbol{u}$ – and of the forest $T \setminus F$ – with capacities given by the capacities given by the capacity vector $\boldsymbol{u}^T$ that, as we will see shortly, can be computed in $\widetilde{O}(m)$ time.

To compute the capacity vector $\boldsymbol{u}^T$, we just note that for any $e \in E_T$, $u^T(e) = u^T[G](e)$, where $u^T[G](e)$ is defined in section 5.4.1. Thus we can use the algorithm from Lemma 5.4.2 to compute all the $u^T(e)$ in $\widetilde{O}(m)$ time. □

Now, we define $\mathcal{H}[j]$ to be the family of all the graphs $H(T,F)$ arising from all the possible choices of a spanning tree T of G and of a subset F of edges of T such that $|F| \leq j$.

5.5.2 Obtaining an $(\widetilde{O}(\log n), \mathcal{H}[\widetilde{O}(\frac{m \log U}{t})])$-decomposition of G

The way the desired t-sparse $(\widetilde{O}(\log n), \mathcal{H}[\widetilde{O}(\frac{m \log U}{t})])$-decomposition of G will be constructed is by employing the multiplicative-weights-update-based approach from Section 2.8. To this end, consider a $(\mathcal{F}[j], G)$-system with $\mathcal{F}[j]$ consisting of flows f_H, for each $H = H(T,F)$ that is in $\mathcal{H}[j]$. Here, the flow f_H is a flow resulting from identity embedding of H into G, i.e., f_H puts $u^T(e)$ units of flow on each $e \in T \setminus F$ and u_e units of flow on each $e \in E[T](F)$.

Clearly, having any feasible solution $\{\lambda_f\}_{f \in \mathcal{F}[j]}$ to an α-relaxed $(\mathcal{F}[j], G)$-system gives us an $(\alpha, \mathcal{H}[j])$-decomposition, as G is embeddable into each $H(T,F)$. Also, if this solution has at most t λ_fs non-zero then this corresponding $(\alpha, \mathcal{H}[j])$-decomposition will be t-sparse. Thus, our goal is to obtain such a solution for α being $\widetilde{O}(\log n)$ and $j = \widetilde{O}(\frac{m \log U}{t})$, sufficiently fast.

To achieve this goal we will design a nearly-linear time algorithm that for any length vector $\boldsymbol{l}$ in G finds a graph $H_{\boldsymbol{l}} = (V, E_{H_{\boldsymbol{l}}}, \boldsymbol{u}^{H_{\boldsymbol{l}}})$ that belongs to the family $\mathcal{H}[\widetilde{O}(\frac{m \log U}{t})]$ and:

(i) $\sum_e l_e u_e^{H_{\boldsymbol{l}}} \leq \alpha l(G)$, where $l(G) := \sum_e l_e u_e$ is the volume of G with respect to $\boldsymbol{l}$;

(ii) $|\kappa(H_{\boldsymbol{l}})| \geq \frac{4\alpha m \log m}{t}$, where $\kappa(H_{\boldsymbol{l}})$ is the set of edges e of $H_{\boldsymbol{l}}$ with $\frac{u_e^{H_{\boldsymbol{l}}}}{u_e} \geq \frac{\rho(H_{\boldsymbol{l}})}{2}$ and $\rho(H_{\boldsymbol{l}}) := \max_{e'} \frac{u_{e'}^{H_{\boldsymbol{l}}}}{u_{e'}}$.

To understand the motivation behind the two above conditions, one needs to note that if $\boldsymbol{w}$ is some weight vector and one takes the length vector $\boldsymbol{l}$ to be $l_e := \frac{w_e}{u_e}$ for each e, then the flow f_{H_l} corresponding to $H_{\boldsymbol{l}}$ as above constitutes a valid answer of an α-oracle for the $(\mathcal{F}[\widetilde{O}(\frac{m \log U}{t})], G)$-system with respect to the weights $\boldsymbol{w}$. To see why this is the case, we just need to note that the condition (i) asserts that

$$\sum_e w_e \mathsf{cong}_{f_{H_l}}(e) = \sum_{e \in E_{H_l}} l_e u_e^{H_l} \leq \alpha \sum_{e \in E} l_e u_e = \alpha \sum_e w_e,$$

where we use the fact that $u_e^{H_l} = f_{H_l}(e)$. Thus, f_{H_l} satisfies the requirements of the oracle definition (cf. Definition 2.8.2). (Interestingly, this oracle never returns **"fail"**– even when the underlying $(\mathcal{F}[\widetilde{O}(\frac{m \log U}{t})], G)$-system is infeasible.)

Furthermore, condition (ii) implies that

$$|\kappa_G(f_{H_l})| = |\kappa(H_{\boldsymbol{l}})| \geq \frac{4\alpha m \log m}{t},$$

where we again used the fact that $u_e^{H_l} = f_{H_l}(e)$ and thus $\rho_G(f_{H_l}) = \rho(H_{\boldsymbol{l}})$ and $\kappa_G(f_{H_l}) = \kappa(H_{\boldsymbol{l}})$. This mean that the tightness of the resulting oracle is at least $\frac{4\alpha m \log m}{t}$.

Now, by employing the multiplicative-weights-update routine from Section 2.8 with $\delta = 1$ and the above oracle we can obtain a feasible solution $\{\lambda_f\}_f$ to the α-relaxed $(\mathcal{F}[\widetilde{O}(\frac{m \log U}{t})], G)$-system. Moreover, Lemma 2.8.9 allows us to conclude that this routine will run in total $\widetilde{O}(mt)$ time and produce a solution that has at most t non-zero λ_fs. So, in the light of the above, we can just focus on designing a nearly-linear time algorithm that finds for any length function $\boldsymbol{l}$ the graph $H_{\boldsymbol{l}}$ that satisfies conditions (i) and (ii) above with $\alpha = \widetilde{O}(\log n)$.

Construction of the Graph $H_{\boldsymbol{l}}$

Let us fix the length vector $\boldsymbol{l}$ throughout this section. Our construction of the graph $H_{\boldsymbol{l}}$ is based on first finding a tree $T_{\boldsymbol{l}}$ that satisfies the condition (i) (for some $\alpha = \widetilde{O}(\log n)$, with respect to capacity vector $\boldsymbol{u}^T$) and then computing a subset $F_{\boldsymbol{l}}$ of $\widetilde{O}(\frac{m \log U}{t})$ edges such that taking $H_{\boldsymbol{l}} = H(T_{\boldsymbol{l}}, F_{\boldsymbol{l}})$ satisfies both conditions and the resulting graph is in $\mathcal{H}[\widetilde{O}(\frac{m \log U}{t})]$.

To find the tree $T_{\boldsymbol{l}}$ we will use the low-stretch spanning tree construction from Theorem 2.6.3.

Lemma 5.5.2. *We can find in $\widetilde{O}(m)$ time a spanning tree $T_{\boldsymbol{l}}$ of G such that if we impose capacities $\boldsymbol{u}^{T_{\boldsymbol{l}}}$ on the edges of $T_{\boldsymbol{l}}$ then $l(T_{\boldsymbol{l}}) = \sum_e l_e u_e^{T_{\boldsymbol{l}}} \leq 2\overline{\alpha} l(G)$, for some $\overline{\alpha}$ being $\widetilde{O}(\log n)$.*

Proof. Consider some spanning tree $T = (V, E_T)$ of G with capacities $\boldsymbol{u}^T$ on its edges.

Note that

$$\sum_f l_f u_f^T = \sum_{f \in E_T} l_f \sum_{e \in E, f \in \mathsf{path}_T(e)} u_e = \sum_{e \in E} l(\mathsf{path}_T(e)) u_e = \sum_{e \in E} \mathsf{stretch}_T^{\boldsymbol{l}}(e) l_e u_e,$$

where we recall from Section 2.6 that the stretch $\mathsf{stretch}^{\boldsymbol{l}}_T(e)$ of an edge e is the ratio of the distance $l(\mathsf{path}_T(e))$ between the endpoints of e in T and the length l_e of the edge e.

Therefore, we see that to establish the lemma it is sufficient to find in $\widetilde{O}(m)$ time a tree T such that

$$\sum_e \mathsf{stretch}^{\boldsymbol{l}}_T(e) l_e u_e \leq 2\overline{\alpha} l(G).$$

To this end, let us follow a technique used in [8] and define a multigraph $\overline{G}$ on vertex set V that contains $d_e := 1 + \lfloor \frac{l_e u_e |E|}{l(G)} \rfloor$ copies of each edge $e \in E$. Note that the total number of edges of $\overline{G}$ – when counting multiple copies separately – is

$$\sum_e d_e \leq |E| + \sum_e \frac{l_e u_e |E|}{l(G)} \leq 2|E| = 2m.$$

Now, we use the algorithm[8] from Theorem 2.6.3 on $\overline{G}$ with weights $w_e = l_e$ for each e, and obtain in $\widetilde{O}(\sum_e d_e) = \widetilde{O}(m)$ time a spanning tree $T_{\boldsymbol{l}}$ of $\overline{G}$ – that also must be a spanning tree of G – such that:

$$\frac{\sum_e \mathsf{stretch}^{\boldsymbol{l}}_{T_{\boldsymbol{l}}}(e) d_e}{\sum_e d_e} \leq \overline{\alpha},$$

for an $\overline{\alpha}$ being $\widetilde{O}(\log n)$.

But, by the fact that for any e

$$d_e \geq \frac{l_e u_e |E|}{l(G)} \geq \frac{l_e u_e \sum_{e'} d_{e'}}{2l(G)},$$

we get

$$\frac{\sum_e \mathsf{stretch}^{\boldsymbol{l}}_{T_{\boldsymbol{l}}}(e) l_e u_e}{2l(G)} \leq \frac{\sum_e \mathsf{stretch}^{\boldsymbol{l}}_{T_{\boldsymbol{l}}}(e) d_e}{\sum_{e'} d_{e'}} \leq \overline{\alpha}.$$

This means that

$$l(T_{\boldsymbol{l}}) = \sum_f l_f u_f^{T_{\boldsymbol{l}}} = \sum_{e \in E} \mathsf{stretch}^{\boldsymbol{l}}_{T_{\boldsymbol{l}}}(e) l_e u_e \leq 2\overline{\alpha} l(G)$$

with $\overline{\alpha} = \widetilde{O}(\log n)$, as desired. □

So, we see that by taking $T_{\boldsymbol{l}}$ as in the above lemma we obtain a graph in $\mathcal{H}[0] \subseteq \mathcal{H}[\widetilde{O}(\frac{m \log U}{t})]$ that satisfies condition (i). However, this graph may still violate the remaining condition (ii). To illustrate our way of alleviating this shortcoming of $T_{\boldsymbol{l}}$, consider a hypothetical situation in which $\kappa(T_{\boldsymbol{l}}) = \{e\}$ for some edge e and for all the other edges $f \neq e$ of $T_{\boldsymbol{l}}$ we have $\frac{u_f^{T_{\boldsymbol{l}}}}{u_f} = \rho$ for some $\rho << \rho(T_{\boldsymbol{l}}) = \frac{u_e^{T_{\boldsymbol{l}}}}{u_e}$. Clearly, in this

[8] Technically, the algorithm of [1] is designed for simple graphs, but it is straight-forward to adapt it to work on multigraphs with a running time being nearly-linear in the total number of edges.

case T_l is a bad candidate for the graph H_l (at least when t is not very large) as it severely violates condition (ii).

However, the key thing to notice in this situation is that if – instead of $H_l = T_l$ – we consider the graph $H_l = H(T_l, F)$ with $F = \{e\}$ then for all the edges f of T_l other than e we have $\frac{u_f^{T_l}}{u_f} = \rho$. Furthermore, for each edge $f \in E[T_l](F) = E[T_l](\{e\})$ it is the case that $\frac{u_f^{T_l}}{u_f} = \frac{u(f)}{u(f)} = 1 \leq \rho$. This means that $\rho(H_l) = \rho$ and thus $|\kappa(H_l)| \geq n-2$, which makes H_l satisfy condition (ii) for even very small values of t.

We see, therefore, that in the above, hypothetical, case by adding only one bottlenecking edge e to F and considering the graph $H_l = H(T_l, F)$ – instead of $H_l = T_l$ – we managed to make the size of the set $\kappa(H_l)$ really large. It turns out that we can always make the above approach work. As the following lemma shows, we can always get the size of $\kappa(H_l)$ to be at least $\frac{4(2\overline{\alpha}+1)m}{t}$ while fixing in the above manner only $\widetilde{O}(\frac{m \log U}{t})$ edges of the tree T_l.

Lemma 5.5.3. *We can construct in $\widetilde{O}(m)$ time a graph $H_l = H(T_l, F_l)$, for some $F_l \subseteq E_{T_l}$ with $|F_l| = \widetilde{O}(\frac{m \log U}{t})$, that satisfies the conditions* (i) *and* (ii) *with α equal to $2\overline{\alpha} + 1$.*

Proof. First, we prove that no matter what is our choice of the set F_l the graph $H_l = H(T_l, F_l)$ satisfies condition (i) for $\alpha = (2\overline{\alpha} + 1)$. To this end, note that

$$l(H_l) = \sum_{e \in (E_{T_l} \setminus F_l)} l_e u_e^{T_l} + \sum_{e \in E[T_l](F_l)} l_e u_e \leq l(T_l) + l(G) \leq (2\overline{\alpha} + 1) l(G).$$

We proceed to finding the subset F_l of edges of the tree T_l that will make $H_l = H(T_l, F_l)$ satisfy also the condition (ii). Let us define for $0 \leq j \leq \lfloor \log mU \rfloor$ $F_j(T_l)$ to be the set of edges e of T_l such that $\frac{mU}{2^{j+1}} \leq \frac{u_e^{T_l}}{u_e} < \frac{mU}{2^j}$.

Note that for any edge e of T_l $1 \leq \frac{u_e^{T_l}}{u_e} \leq mU$ – the first inequality follows since e itself has to be routed in T_l over e, and the second one since in the worst case all m edges will be routed in T_l over e. Therefore, the union $\bigcup_{j=0}^{\lfloor \log mU \rfloor} F_j(T_l)$ of all the $F_j(T_l)$s partitions the whole set E_{T_l} of the edges of the tree T_l.

Now, let us take j^* to be the smallest j such that

$$\sum_{j'=j}^{\lfloor \log mU \rfloor} |F_{j'}(T_l)| \leq \frac{4(2\overline{\alpha} + 1)m(\lfloor \log mU \rfloor + 1)}{t}.$$

In other words, j^* is the smallest j such that we could afford to take as F_l the union of all the sets $F_{j^*}(T_l), \ldots, F_{\lfloor \log mU \rfloor}(T_l)$ and still have the size of F_l not exceed the desired cardinality bound of

$$\frac{4(2\overline{\alpha} + 1)m(\lfloor \log mU \rfloor + 1)}{t} = \widetilde{O}(\frac{m \log U}{t}).$$

Note that we can assume that $j^* > 0$. Otherwise, we could just afford to take

F_l to be the union of all the sets $F_j(T_l)$ i.e. $F_l = E_{T_l}$. This would mean that $H_l = H(T_l, E_{T_l})$ is just the graph G itself and thus $\rho(H_l) = 1$ and $|\kappa(H_l)| = m$, which would satisfy the condition (ii).

Once we know that $j^* > 0$, the definition of j^* implies that

$$\sum_{j'=j^*-1}^{\lfloor \log mU \rfloor} |F_{j'}(T_l)| > \frac{4(2\overline{\alpha}+1)m(\lfloor \log mU \rfloor + 1)}{t}.$$

However, since this sum has $j^*+2 \leq \lfloor \log mU \rfloor + 1$ summands, pigeon-hole principle asserts existence of $j^* \leq \overline{j} \leq \lfloor \log mU \rfloor$ such that

$$|F_{\overline{j}-1}(T_l)| \geq \frac{4(2\overline{\alpha}+1)m}{t}.$$

Now, we define F_l to be the union $\bigcup_{j=\overline{j}}^{\lfloor \log mU \rfloor} F_j(T_l)$. By the fact that $j^* \leq \overline{j}$ we know that the size of such F_l obeys the desired cardinality bound. Furthermore, we claim that if we take $H_l = H(T_l, F_l)$ with such choice of F_l then $|\kappa(H_l)|$ is large enough.

To see this, note that the fact that $\frac{u_e^{H_l}}{u_e} = 1$ for all edges e in $E[T_l](F_l)$ implies that $\rho(H_l)$ is at most $\frac{2^{\overline{j}-1}}{mU}$. But this means that all the edges from $F_{\overline{j}-1}(T_l)$ are in $\kappa(H_l)$ and thus

$$|\kappa(H_l)| \geq |F_{\overline{j}-1}(T_l)| \geq \frac{4(2\overline{\alpha}+1)m}{t},$$

as desired.

Now, to conclude the proof, we notice that Lemma 5.5.2 and Lemma 5.5.1 imply that the graph H_l as above can indeed be constructed in $\widetilde{O}(m)$ time – to find the set F_l we just sort the edges of T_l according to $\frac{u_e^{T_l}}{u_e}$ and choose the appropriate value of $\overline{j}$. □

In the light of our above considerations and Lemma 5.5.3 we obtain the following corollary.

Corollary 5.5.4. *A t-sparse $(\overline{\alpha}', \mathcal{H}[\widetilde{O}(\frac{m \log U}{t})])$-decomposition of G can be found in $\widetilde{O}(tm)$ time for some $\overline{\alpha}'$ being $\widetilde{O}(\log n)$.*

5.5.3 Obtaining an $(\widetilde{O}(\log n), \mathcal{G}_V[\widetilde{O}(\frac{m \log U}{t})])$-decomposition of G

We proceed now to transforming the $(\widetilde{O}(\log n), \mathcal{H}[\widetilde{O}(\frac{m \log U}{t})])$-decomposition obtained in the previous section into the $(\widetilde{O}(\log n), \mathcal{G}_V[\widetilde{O}(\frac{m \log U}{t})])$-decomposition that we want to obtain.

To this end, let us call G' an *almost-j-tree* if it is a union of a tree and of an arbitrary graph on at most j vertices. Now, let us consider a graph $H(T, F)$ for some choice of the spanning tree T of G and of a subset F of edges of T. A property of this graph that we will find useful is that the cut-flow structure of $H(T, F)$ is similar

to a cut-flow structure of an almost-$O(|F|)$-tree $G(T,F)$ and that furthermore, this $G(T,F)$ can be found efficiently.

Lemma 5.5.5. *For any spanning tree $T=(V,E_T)$ of G and $F \subseteq E_T$, we can find in $\widetilde{O}(m)$ time an almost-$O(|F|)$-tree $G(T,F)$ such that the graph $H(T,F)$ is embeddable into $G(T,F)$ and $G(T,F)$ is 3-embeddable into $H(T,F)$. Furthermore, the capacity ratio of $G(T,F)$ is at most mU.*

Proof. Consider some edge $e=(v,v')$ from $E[T](F)$. Let us define $v^1(e)$ (resp. $v^2(e)$) to be the vertex that corresponds to the first (resp. last) moment we encounter an endpoint of an edge from F while moving along the path $\mathsf{path}_T(e)$ from v to v'. Also, let us denote by $\mathsf{path}^1_T(e)$ (resp. $\mathsf{path}^2_T(e)$) the fragment of $\mathsf{path}_T(e)$ between v and $v^1(e)$ (resp. between v' and $v^2(e)$).

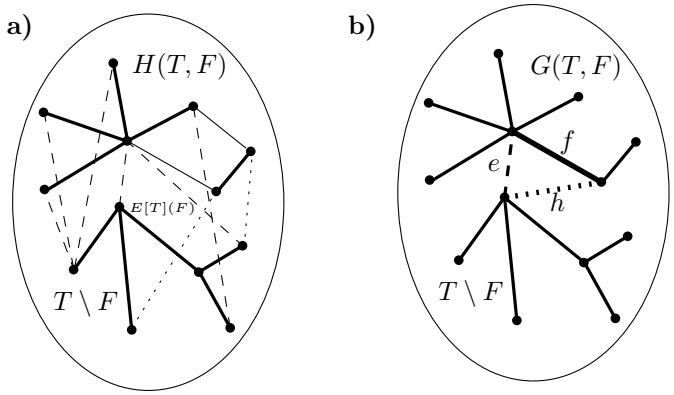


Figure 5-4: **a)** An example of a graph $H(T,F)$ from Figure 5-3. **b)** The corresponding graph $G(T,F)$ with three projected edges e, f, and h. The set $\mathsf{Proj}(e)$ consist of edges of $H(T,F)$ marked as dashed, the edges of the set $\mathsf{Proj}(f)$ are solid, and $\mathsf{Proj}(h)$ contains all the edges that are dotted.

We define the graph $G(T,F)=(V,E',\boldsymbol{u}')$ as follows. We make the edge set E' of $G(T,F)$ to consist of the edges from $E_T \setminus F$ and of an edge $f=(w,w')$ for each w, w' such that $(w,w')=(v^1(e),v^2(e))$ for some edge $e \in E[T](F)$. We will call such an edge f *projected* and define $\mathsf{Proj}(f)$ to be the set of all $e \in E[T](F)$ with $(v^1(e),v^2(e))=(w,w')$. Now, we set the capacity u'_e in $G(T,F)$ to be:

$$u'_e := \begin{cases} 2u^T_e & \text{if } e \in E_T \setminus F; \\ \sum_{e' \in \mathsf{Proj}(e)} u_{e'} & \text{otherwise (i.e. if } e \text{ is projected).} \end{cases}$$

See Figure 5-4 for an example of a graph $H(T,F)$ and the graph $G(T,F)$ corresponding to it.

To see that this construction of $G(T,F)$ can be performed quickly, note that by Lemma 5.5.1 we can find the set $E[T](F)$ and capacity vector $\boldsymbol{u}^T$ in $\widetilde{O}(m)$ time. Moreover, by employing a simple adaptation of the divide-and-conquer approach that

was used in Lemma 5.4.2 to compute $\boldsymbol{u}^T[D]$, we can compute all $v^1(e)$, $v^2(e)$ for each $e \in E[T](F)$ also in $\widetilde{O}(m)$ time.

By noting that if a vertex v is equal to $v^1(e)$ or $v^2(e)$ for some edge e then v has to be an endpoint of an edge in F, we conclude that the subgraph of $G(T,F)$ induced by projected edges is supported on at most $2|F|$ vertices. This means that $G(T,F)$ – as a connected graph being a union of a forest $E_T \setminus F$ and a graph on $2|F|$ vertices – is an almost-$2|F|$-tree. Furthermore, note that $u_e^T \leq m \max_{e' \in E} u_{e'}$ for each edge e and that the capacity u'_f of an projected edge f is upperbounded by $\sum_{e' \in E} u_{e'} \leq m \max_{e' \in E} u_{e'}$. So, the fact that no edge e in $G(T,F)$ has its capacity smaller than u_e implies that the capacity ratio of $G(T,F)$ is at most mU.

To relate the cut-flow structure of $H(T,F)$ and $G(T,F)$, one can embed $H(T,F)$ into $G(T,F)$ by embedding each edge $e \in E_T \setminus F$ of $H(T,F)$ into its counterpart edge in $G(T,F)$ and by embedding each edge $e \in E[T](F)$ by routing the corresponding flow of u_e units along the path formed from paths $\mathsf{path}_T^1(e)$ and $\mathsf{path}_T^2(e)$ connected through the projected edge $(v^1(e), v^2(e))$. It is easy to see that our definition of $\boldsymbol{u}'$ ensures that this embedding does not overflow any capacities of edges of $G(T,F)$.

On the other hand, to 3-embed $G(T,F)$ into $H(T,F)$, we embed each non-projected edge of $G(T,F)$ into the same edge in $H(T,F)$. Moreover, we embed each projected edge f by splitting the corresponding $u'_f = \sum_{e \in \mathsf{Proj}(f)} u_e$ units of flow into $|\mathsf{Proj}(f)|$ parts. Namely, for each $e \in \mathsf{Proj}(f)$, we route u_e units of this flow along the path constituted by paths $\mathsf{path}_T^1(e)$, $\mathsf{path}_T^2(e)$ and the edge e. Once again, it is not hard to convince oneself that such a flow does not overflow the capacities of edges of $H(T,F)$ by a factor of more than three. The lemma follows. □

It is easy to see that every j-tree is an almost-j-tree, however the converse does not necessarily hold – one can consider an almost-2-tree corresponding to a cycle. Fortunately, the following lemma proves that every almost-j-tree G is close – with respect to its cut structure – to some $O(j)$-tree $\overline{G}$ and this $\overline{G}$ can be found efficiently.

Lemma 5.5.6. *Let $G' = (V', E', \boldsymbol{u}')$ be an almost-j-tree, we can obtain in $\widetilde{O}(|E'|)$ time an $O(j)$-tree $\overline{G} = (V', \overline{E}, \overline{\boldsymbol{u}})$ such that G' is embeddable into $\overline{G}$ and $\overline{G}$ is 3-embeddable into G'. Furthermore, the capacity ratio of $\overline{G}$ is at most twice the capacity ratio of G'.*

Proof. Let us assume first that G' does not have vertices of degree one - we will deal with this assumption later. Let $W \subset V'$ be the set of all vertices of degree two in G'. It is easy to see that we can find in $O(|E'|)$ time a collection of edge-disjoint paths $p_1, \ldots, p_k$ covering all vertices in W such that in each p_i all the internal vertices are from W and the two endpoints $v^1(p_i)$ and $v^2(p_i)$ of p_i are not in W, i.e. they have degree at least three in G'.

We construct $\overline{G}$ by first taking the subgraph H' of G' induced by the vertex set $V' \setminus W$ and an empty forest $\overline{F}$. Next, for each path p_i, we repeat the following. Let $e_{\min}(p_i)$ be the edge of p_i that has minimal capacity u'_e among all the edges e of p_i. We add to both $\overline{G}$ and $\overline{F}$ the path p_i with $e_{\min}(p_i)$ removed . We set the capacities $\overline{\boldsymbol{u}}$ of these added edges to be equal to their capacity in G' increased by the capacity $u'_{e_{\min}(p_i)}$ of $e_{\min}(p_i)$. Finally, we add an edge $(v^1(p_i), v^2(p_i))$ to $\overline{G}$ with capacity equal

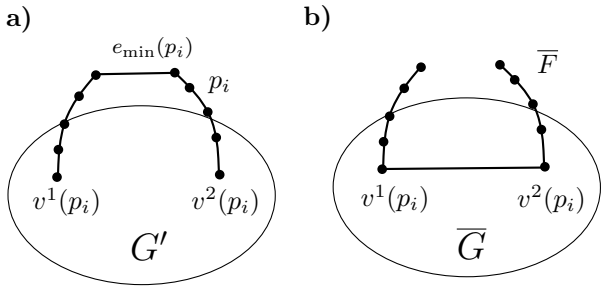


Figure 5-5: **a)** An example of a path p_i found in G'. **b)** Transformed path p_i in the resulting graph $\overline{G}$.

to $u'_{e_{\min}(p_i)}$. See Figure 5-5. Clearly, such a construction can be performed in $O(|E'|)$ time and the capacity ratio of $\overline{G}$ can be at most twice the capacity ratio of G'.

We claim that the graph $\overline{G}$ obtained above is a $(3j-2)$-tree with $\overline{F}$ being its envelope and its core being the subgraph of $\overline{G}$ induced by vertex set $V' \setminus W$ (note that this subgraph consists of H' and all the edges $\{(v^1(p_i), v^2(p_i))\}_i$). One can see that we only need to argue that $|V' \setminus W| \leq 3j+2$, the rest of the claim follows immediately from the above construction. To this end, note that since G' is an almost-j-tree at least $|V'|-j$ of its vertices is incident only to edges of the underlying spanning tree on V'. This means that the total sum of degrees of these vertices in G' is at most $2(|V'|-1)$. But there is no vertices in G' with degree smaller than two, thus a simple calculation shows that at most $2(j-1)$ of them can have degree bigger than two. We can conclude therefore that $|W| \geq |V'|-j-2(j-1)$ and $|V' \setminus W| \leq 3j-2$, as desired.

Now, to see that G' embeds into $\overline{G}$, we note that H' is already contained in $\overline{G}$. So, we only need to take care of the paths $\{p_i\}_i$. For each path p_i, $\overline{G}$ already contains all its edges except $e_{\min}(p_i)$. But this edge can be embedded into $\overline{G}$ by just routing the corresponding flow from one of its endpoints, along the fragment of path leading to $v^1(p_i)$, then through the edge $(v^1(p_i), v^2(p_i))$ and finally back along the path to the other endpoint. It is easy to verify that our way of setting up capacities ensures that this routing will not overflow any of them.

Similarly, to 3-embed $\overline{G}$ into G' we first embed H' into G' via identity embedding. Subsequently, for each path p_i, G' already contains all its edges, so we can use identity embedding for them as well. Furthermore, to embed the remaining edges $\{(v^1(p_i), v^2(p_i))\}_i$ we can just route, for each i, the corresponding flow from $v^1(p_i)$ to $v^2(p_i)$ along the path p_i. Once again, one can convince oneself that by definition of $e_{\min}(p_i)$ the resulting embedding does not overflow any capacities in G' by a factor of more than three.

To conclude the proof, it remains to explain how to deal with graph G' having vertices of degree one. In this case one may preprocess G' as follows. We start with an empty forest $\overline{F}'$ and as long as there is a degree one vertex in G' we remove it – together with the incident edge – from G' and we add both the vertex and the edge to $\overline{F}'$. Once this procedure finishes the resulting graph G' will not have vertices

of degree one and still will be an almost-j-tree. Thus, we can use our algorithm described above and then just add $\overline{F}'$ to the $(3j-2)$-tree that the algorithm outputs. It is easy to see that the resulting graph will be still a $(3j-2)$-tree – with $\overline{F}'$ being part of its envelope – having all the desired properties. □

Note that the above lemmas can be combined to produce – in $\widetilde{O}(m)$ time – for a given graph $H(T,F)$ an $O(|F|)$-tree that preserves the cut-flow structure of $H(T,F)$ up to a factor of nine. This allows us to prove Theorem 5.3.6.

Proof of Theorem 5.3.6. By Corollary 5.5.4 we can compute in $\widetilde{O}(tm)$ time a t-sparse $(\overline{\alpha}', \mathcal{H}[\widetilde{O}(\frac{m \log U}{t})])$-decomposition $\{(\lambda^i, H(T^i, F^i))\}_i$ of G with $\overline{\alpha}'$ being $\widetilde{O}(\log n)$. Now, consider a convex combination $\{(\lambda^i, \overline{G}^i)\}_i$ with each $\overline{G}^i$ being the $\widetilde{O}(\frac{m \log U}{t})$-tree produced by applying Lemma 5.5.5 and then Lemma 5.5.6 to the graph $H(T^i, F^i)$. Clearly, we can obtain this combination in $\widetilde{O}(tm)$ time.

We claim that this combination is a t-sparse $(\widetilde{O}(\log n), \mathcal{G}[\widetilde{O}(\frac{m \log U}{t})])$-decomposition of G. Obviously, $\sum_i \lambda^i = 1$ and G is embeddable into each $\overline{G}^i$ since G is embeddable into each $H(T^i, F^i)$ and each $H(T^i, F^i)$ is embeddable into corresponding $\overline{G}^i$. Similarly, by composing the 9-embedding of $\overline{G}^i$ into $H(T^i, F^i)$ and the identity embedding of $H(T^i, F^i)$ into G (cf. embeddings $\{f_j\}_j$ in Definition 5.3.1), we get that $\{(\lambda^i, \overline{G}^i)\}_i$ satisfies all the requirements of a t-sparse $(9\overline{\alpha}', \mathcal{G}_V[\widetilde{O}(\frac{m \log U}{t})])$-decomposition. Furthermore, by Lemma 5.5.5 and Lemma 5.5.6 the capacity ratio of each $\overline{G}^i$ is at most $2mU$. The theorem follows. □

5.6 Proof of Theorem 5.3.7

Finally, we are ready to prove the main theorem of this chapter – Theorem 5.3.7. The heart of the algorithm that will prove this theorem is the procedure Find_Trees(G, t, j) described in Figure 5-6. On the high level, this procedure given an input graph G aims at finding a j-tree that α-preserves the cuts of G with some probability p, where both α and p depend on the parameters t and j. This is done by first cut sparsifying G (cf. Corallary 2.6.2) to reduce the number of its edges, then using the decomposition of Theorem 5.3.6 to represent the resulting graph as a convex combination of t simpler graphs (j'-trees, for some $j' \geq j$). Next, one of these simpler graphs H is sampled – in the spirit of Fact 5.3.4 – and if H is already a j-tree then it is returned. Otherwise, the procedure recursively finds a j-tree $\overline{G}'$ that approximates the cuts of the core of H – thus $\overline{G}'$ is only defined on the vertex set of this core – and returns a j-tree that approximates the cuts of the whole H – and thus of G – by just adding the envelope of H to $\overline{G}'$. An important detail is that whenever we recurse, we increase the value of parameter t by squaring it. Intuitively, we may employ this more aggressive setting of the value of t since the size of the problem – i.e. the size of the core of H – decreased by a factor of at least t.

To justify our recursive step we prove the following lemma.

Procedure Find_Trees(G, t, j):
  Input : Graph $G = (V, E, \boldsymbol{u})$, parameters $t \geq 1$, and $j \geq 1$
  Output: A j-tree $\overline{G}$ on vertex set V
  Obtain a cut sparsification $\widetilde{G}$ of G as in Corollary 2.6.2 with $\delta = 1$
  Find a t'-sparse $(\widetilde{O}(\log |V|), \mathcal{G}_V[\max\{\frac{|V|}{t}, 1\}])$-decomposition $\{(\lambda^i, \widetilde{G}^i)\}_i$ of $\widetilde{G}$ as in Theorem 5.3.6, where t' is chosen so as to make the decomposition consist of $\max\{\frac{|V|}{t}, 1\}$-trees
  Sample a graph H by choosing $\widetilde{G}^i$ with probability λ^i, as in Fact 5.3.4
  if H is a j-tree **then**
    $\overline{G} \leftarrow H$
  else
    Let H' be the core of H and let F be the envelope of H
    Invoke Find_Trees(H', t^2, j) to obtain a j-tree $\overline{G}'$ on vertex set $V' \subseteq V$ of the graph H'
    Add F to $\overline{G}'$ to get a j-tree $\overline{G}$ on vertex set V
  end
  return $\overline{G}$

Figure 5-6: Procedure Find_Trees(G, t, j)

Lemma 5.6.1. *Let $H = (V, E, \boldsymbol{u})$ be a j'-tree and let $H' = (V', E', \boldsymbol{u})$ be its core. Also, let $\overline{G}' = (V', \overline{E}, \overline{\boldsymbol{u}})$ be a j-tree on vertex set V' that α-preserves the cuts of H' with probability p, for some $\alpha \geq 1$ and $p > 0$. Then the graph $\overline{G}$ being a union of $\overline{G}'$ and the envelope F of H is a j-tree on vertex set V that α-preserves the cuts of H with probability p.*

Proof. Note that for any cut $\emptyset \neq C \subset V$, the capacity $u(C)$ of this cut in H is equal to the capacity $u_{H'}(C)$ of this cut with respect to edges of H' plus its capacity $u_F(C)$ with respect to edges of F. Similarly, the capacity of C in $\overline{G}$ is equal to its capacity $\overline{u}(C)$ in $\overline{G}'$ plus the capacity $u_F(C)$. Therefore, the lemma follows by the fact that $\overline{G}'$ α-preserves the cuts of H' with probability p and that after adding an envelope F to it $\overline{G}'$ is still a j-tree. □

We proceed to analyzing the running time and the quality of cut preservation offered by the j-tree found by the procedure Find_Trees(G, t, j).

Lemma 5.6.2. *For a given graph $G = (V, E, \boldsymbol{u})$ and $j \geq 1$, and $t \geq 2$, Procedure Find_Trees(G, t, j) works in $\widetilde{O}(m + tnT^2 \log U)$ time and returns a j-tree $\overline{G}$ that $(\log^{(1+o(1))T} n)$-preserves the cuts of G with probability $(1/2)^T$, where $T \leq \max\{\lceil \log(\log_t(n/j) + 1) \rceil, 1\}$, $n = |V|$, $m = |E|$, and U is the capacity ratio of G. Moreover, the capacity ratio of $\overline{G}$ is $n^{(2+o(1))T} U$.*

Proof. Let us define n_l to be an upper bound on the number of the vertices of the graph that we are dealing with at the beginning of the l-th recursive call of the procedure. Also, let t_l be the value of t and let U_l be the upper bound on the capacity

ratio of this graph at the same moment. Clearly, $n_0 = n$, $t_0 = t$, and $U_0 = U$. By solving simple recurrence we see that

$$t_l = t^{2^l} \tag{5.3}$$

and thus

$$n_{l+1} \leq \frac{n_l}{t_l} \leq \frac{n}{t^{2^{l+1}-1}}. \tag{5.4}$$

Now, by Theorem 5.3.6 and Corollary 2.6.2, we know that U_{l+1} is $O(n_l^2 U_l)$. Moreover, by the fact that we sparsify G first and by Theorem 5.3.6, we know that in l-th iteration it is sufficient to take $t' = t_l \hat{t}_l$, where $\hat{t}_l$ is $\log^{O(1)} n \log U_l$.

Note that every graph on at most j vertices is a j-tree thus the procedure stops issuing further recursive calls once the number of vertices of G is at most j. So, the number of recursive calls of our procedure is at most $T - 1$, where

$$T := \max\{\lceil \log(\log_t(n/j) + 1) \rceil, 1\}.$$

As a result, the bound on the capacity ratio of $\overline{G}$ is $U_T \leq n^{(2+o(1))^T} U$, as desired.

To bound the total running time of the procedure, consider its modification in which we additionally sparsify the graph H' before passing it to the recursive call. Clearly, running time of such modification can only increase. Note that in this case all but the topmost invocation of the procedure deals from the beginning till the end with sparsified version of the graph. Therefore, the time need to execute l-th recursive call, for $l \geq 1$, is $\widetilde{O}(\hat{t}_l t_l n_l)$. By equations (5.3) and (5.4), this implies that the total time taken by the execution of our procedure is:

$$\begin{aligned} \widetilde{O}(m) + \sum_{l=0}^{T-1} \widetilde{O}(\hat{t}_l t_l n_l) &\leq \widetilde{O}(m + \sum_{l=0}^{T-1} \frac{n t^{2^l} \log U_l}{t^{2^l - 1}}) \\ &\leq \widetilde{O}(m + tn \sum_{l=0}^{T-1} \log U_l) = \widetilde{O}(m + tnT^2 \log U), \end{aligned}$$

where $\widetilde{O}(m)$ is the cost of the initial cut sparsification of G.

To establish the desired bound on the quality of the preservation of cuts of G by $\overline{G}$, we prove that for all $l \geq 0$, if the execution of the procedure made l recursive calls then the graph $\overline{G}$ $(\log^{(1+o(1))(l+1)} n)$-preserves the cuts of G with probability $(1/2)^{l+1}$. Clearly, this claim implies the desired bounds by combining it with the fact that $l \leq T - 1$.

We prove the claim inductively. For $l = 0$ it follows from Corollary 2.6.2, Theorem 5.3.6, and Fact 5.3.4. Now, assume that the claim holds for $l \geq 0$ and we will prove it for $l + 1$. By Corollary 2.6.2, Theorem 5.3.6, and Fact 5.3.4, we know that H is $(\log^{1+o(1)} n)$-preserving the cuts of G with probability $1/2$. So, by our inductive assumption we know that $\overline{G}'$ $(\log^{(1+o(1))(l+1)} n)$-preserves the cuts of H' with probability $(1/2)^{l+1}$. Therefore, by Lemma 5.6.1 we know that $\overline{G}$ also $(\log^{(1+o(1))(l+1)} n)$-preserves

the cuts of H with probability $(1/2)^{l+1}$ which – by the above-mention relation of the cut-structure of H to the cut-structure of G – implies that $\overline{G}$ $(\log^{(1+o(1))(l+2)} n)$-preserves the cuts of H with probability $(1/2)^{l+2}$. So, our claim holds for $l+1$ and thus for all the values of l. This concludes the proof of the lemma. □

Now, proving Theorem 5.3.7 boils down to running the Find_Trees(G, t, j) for the right setting of parameters and sufficiently many times, so as to boost the probability that the obtained collection of j-trees preserves cuts with high probability.

Proof of Theorem 5.3.7. To obtain the desired collection of n^l-trees, we just invoke procedure Find_Trees(G, t, j) on G $(2^{k+1} \ln n)$ times with $t := n^{\frac{1-l}{2^k-1}}$. Note that for this choice of parameters we get that in the statement of Lemma 5.6.2 $T \leq k$.

Therefore, this lemma allows us to conclude that each invocation of the procedure takes $\widetilde{O}(n^{(1+1/k)} \log U)$ time to execute and thus we obtain $(2^{k+1} \ln n)$ n^l-trees $\{G^i\}_i$ in total time of

$$\widetilde{O}(m + 2^k n^{1+\frac{1-l}{2^k-1}} \log U).$$

(We used here the fact that it is sufficient to cut sparsify G only once for all the procedure calls.)

Furthermore, Lemma 5.6.2 implies that the capacity ratio of the obtained trees is at most $n^{(2+o(1))k} U$ and each particular n^l-tree G_i $(\log^{(1+o(1))k} n)$-preserves the cuts of G with probability $(1/2)^k$. However, it is easy to see that this in turn means that the whole collection $\{\overline{G}^i\}_i$ $(\log^{(1+o(1))k} n)$-preserved the cuts of G with probability $(1-(1-(1/2)^k)^{2^{k+1} \ln n}) = (1-1/n^2)$, as desired. The theorem follows.

□

Part II

Beyond Cuts and Flows

Chapter 6

Approximation of the Asymmetric Traveling Salesman Problem

In this chapter, we study the asymmetric traveling salesman problem.[1] We derive a randomized algorithm which delivers a solution within a factor $O(\frac{\log n}{\log \log n})$ of the optimum, with high probability. This improves upon the long-standing approximation barrier of $\Theta(\log n)$.

To achieve this improvement, we combine the traditional polyhedral approaches with randomized rounding procedure that is based on the notion of random spanning trees and exploits their connection to electrical flows and Laplacian systems – cf. Section 2.5.2 and Fact 2.5.2.

6.1 Introduction

The traveling salesman problem (TSP) – the task of finding a minimum cost tour that visits each city in a given set at least once – is one of the most celebrated and extensively studied problems in combinatorial optimization. The book by Lawler *et al.* [98] provides a tour d'horizon of all the concepts and techniques developed in the context of TSP.

The traveling salesman problem comes in two variants. The symmetric version (STSP) assumes that the cost $c(v, u)$ of getting from city v to city u is equal to $c(u, v)$, while the more general asymmetric version (ATSP) that we study here does not make this assumption. Note that since we can replace every arc (u, v) in the tour with the shortest path from u to v, we can assume that the cost function c satisfies the triangle inequality, i.e., we have $c(u, w) \leq c(u, v) + c(v, w)$ for all u, v, and w.

Despite a lot of interest and work, little progress has been made over the last three decades on its approximability in the general metric case. For the symmetric variant, there is a celebrated factor $3/2$ approximation algorithm due to Christofides [41]. This algorithm is based on first finding a minimum cost spanning tree T on V, then

[1]This chapter is based on joint work with Arash Asadpour, Michel Goemans, Shayan Oveis Gharan, and Amin Saberi and contains material from [18].

finding the minimum cost Eulerian augmentation of that tree, and finally shortcutting the corresponding Eulerian walk into a tour. No better approximation algorithm has since been found for the general symmetric metric case. For the asymmetric case, no constant approximation algorithm is known. Frieze *et al.* [69] gave a simple $\log n$-approximation algorithm for ATSP, which was improved slightly by subsequent results down to $2/3 \log n$ [30, 79, 63]. This is in sharp contrast to the best inapproximability result of Papadimitriou and Vempala [112] which shows the nonexistence of an 117/116-approximation algorithm for the ATSP, and of an 220/219-approximation algorithm for the STSP, unless $P = NP$. Whether ATSP can be approximated within a constant factor is a major open question, and so is whether an c-approximation algorithm for STSP can be obtained for a constant $c < 3/2$.

In this chapter, we make progress on this question by presenting an $O(\frac{\log n}{\log \log n})$-approximation algorithm for the ATSP. This approximation factor finally breaks the long-standing $\Theta(\log n)$ barrier mentioned above. Our approach to the asymmetric version has similarities with Christofides' algorithm; we first construct a spanning tree with special properties. Then we find a minimum cost Eulerian augmentation of this tree, and finally, shortcut the resulting Eulerian walk. (Recall that for undirected graphs, being Eulerian means being connected and having even degrees, while for directed graphs it means being (strongly) connected and having the indegree of every vertex equal to its outdegree.)

Our way of bounding the cost of the Eulerian cost augmentation is based on applying a simple flow argument using Hoffman's circulation theorem [121] that shows that if the tree chosen in the first step is "thin" then the cost of the Eulerian augmentation is within a factor of the "thinness" of the (asymmetric) Held-Karp linear programming (LP) relaxation value ($\mathrm{OPT}_{\mathrm{HK}}$) [82]. This flow argument works irrespectively of the actual directions of the (directed) arcs corresponding to the (undirected) edges of the tree. Roughly speaking, a *thin* tree with respect to the optimum solution $\boldsymbol{x}^*$ of the Held-Karp relaxation is a spanning tree that, for every cut, contains a small multiple (the *thinness*) of the corresponding value of $\boldsymbol{x}^*$ in this cut when the direction of the arcs are disregarded.

A key step of our algorithm is to find a thin tree of small cost compared to the LP relaxation value $\mathrm{OPT}_{\mathrm{HK}}$. For this purpose, we employ a sampling procedure that produces a random spanning tree (cf. Section 2.5.2) with respect to weights $\{\lambda_e\}_e$ that are chosen so as to make the probabilities of each edge appearing in this tree to (approximately) agree with marginal probabilities obtained from the symmetrized LP solution (scaled by $1 - 1/n$). To make this procedure efficient, we develop a simple iterative algorithm for (approximately) computing these λ_e's efficiently.

The main motivation behind choosing such a sampling procedure is that the events corresponding to edges being present in the sampled tree are negatively correlated. This means that the upper tail of well-known Chernoff bound for the independent setting still holds, see Panconesi and Srinivasan [111]. The proof of the $O(\frac{\log n}{\log \log n})$-thinness of the sampled tree is based on combining this tail bound with "union-bounding over cuts" technique of Karger [86].

The high level description of our algorithm can be found in Figure 6-1. The proof of our main Theorem 6.6.4 also gives a more formal overview of the algorithm.

INPUT: A set V consisting of n points and a cost function $c : V \times V \to \mathbb{R}^+$ satisfying the triangle inequality.

OUTPUT: $O(\frac{\log n}{\log \log n})$-approximation to the asymmetric traveling salesman problem on V.

ALGORITHM:

1. Solve the Held-Karp LP relaxation of the ATSP instance to get an optimum extreme point solution $\boldsymbol{x}^*$. [See LP (6.1).] Define $\boldsymbol{z}^*$ by (6.5); $\boldsymbol{z}^*$ can be interpreted as the marginal probabilities on the edges of a probability distribution on spanning trees.
2. Sample $\Theta(\ln n)$ spanning trees $T_1, T_2, \ldots$ from a distribution that corresponds to a random spanning tree that approximately preserves the marginal probabilities imposed by $\boldsymbol{z}^*$. Let T^* be the tree with minimum (undirected) cost among all the sampled trees. [See Sections 6.4 and 6.5.]
3. Orient each edge of T^* so as to minimize its cost. Find a minimum cost integral circulation that contains the oriented tree $\vec{T}^*$. Shortcut this multigraph and output the resulting tour.

Figure 6-1: An $O(\frac{\log n}{\log \log n})$-approximation algorithm for the ATSP.

6.2 Notation

Before describing our approximation algorithm for ATSP in details, we need to introduce some notation. Throughout this chapter, we use $a = (u, v)$ to denote the arc (directed edge) from u to v and $e = \{u, v\}$ for an undirected edge. Also we use A (resp. E) for the set of arcs (resp. edges) in a directed (resp. undirected) graph.

For a given function $f : A \to \mathbb{R}$, the cost of f is defined as follows:

$$c(f) := \sum_{a \in A} c(a) f(a).$$

For a set $S \subseteq A$, we define

$$f(S) := \sum_{a \in S} f(a).$$

We use the same notation for a function defined on the edge set E of an undirected

graph. For $U \subseteq V$, we also define the following sets of arcs:

$$\begin{aligned} \delta^+(U) &:= \{a = (u,v) \in A : u \in U, v \notin U\}, \\ \delta^-(U) &:= \delta^+(V \backslash U) \\ A(U) &:= \{a = (u,v) \in A : u \in U, v \in U\}. \end{aligned}$$

Similarly, for an undirected graph $G = (V, E)$, $\delta(U)$ denotes the set of edges with exactly one endpoint in U, and $E(U)$ denotes the edges entirely within U, i.e. $E(U) = \{\{u, v\} \in E : u \in U, v \in U\}$.

6.3 The Held-Karp Relaxation

Given an instance of ATSP corresponding to the cost function $c : V \times V \to \mathbb{R}^+$, we can obtain a lower bound on the optimum value by considering the following linear programming relaxation defined on the complete bidirected graph with vertex set V:

$$\min \sum_a c(a)x_a \tag{6.1}$$

$$\text{s.t. } \boldsymbol{x}(\delta^+(U)) \geq 1 \qquad \forall U \subset V, \tag{6.2}$$

$$\boldsymbol{x}(\delta^+(v)) = \boldsymbol{x}(\delta^-(v)) = 1 \qquad \forall v \in V, \tag{6.3}$$

$$x_a \geq 0 \qquad \forall a.$$

This relaxation is known as the Held-Karp relaxation [82] and its optimum value, which we denote by OPT_{HK}, can be computed in polynomial-time (either by the ellipsoid algorithm or by reformulating it as an LP with polynomially-bounded size). Observe that (6.3) implies that any feasible solution $\boldsymbol{x}$ to the Held-Karp relaxation satisfies

$$\boldsymbol{x}(\delta^+(U)) = \boldsymbol{x}(\delta^-(U)), \tag{6.4}$$

for any $U \subset V$.

Let $\boldsymbol{x}^*$ denote an optimum solution to this LP (6.1); thus $c(\boldsymbol{x}^*) = \text{OPT}_{\text{HK}}$. We can assume that $\boldsymbol{x}^*$ is an extreme point of the corresponding polytope. We first make this solution symmetric and slightly scale it down by setting

$$\boldsymbol{z}^*_{\{u,v\}} := \frac{n-1}{n}(x^*_{uv} + x^*_{vu}). \tag{6.5}$$

Let A denote the support of $\boldsymbol{x}^*$, i.e. $A = \{(u,v) : x^*_{uv} > 0\}$, and E the support of $\boldsymbol{z}^*$. For every edge $e = \{u, v\}$ of E, we can define its cost as $\min\{c(a) : a \in \{(u,v),(v,u)\} \cap A\}$; with the risk of overloading the notation, we denote this new cost of this edge e by $c(e)$. This implies that $c(\boldsymbol{z}^*) < c(\boldsymbol{x}^*)$.

The main purpose of the scaling factor in the definition (6.5) is to obtain a vector $\boldsymbol{z}^*$ which belongs to the *spanning tree polytope* P of the graph (V, E), i.e. $\boldsymbol{z}^*$ can be viewed as a convex combination of incidence vectors of spanning trees, see Lemma 6.3.1. In fact, $\boldsymbol{z}^*$ even belongs to the relative interior of P.

Lemma 6.3.1. *The vector $\boldsymbol{z}^*$ defined by (6.5) belongs to the relative interior of the spanning tree polytope P.*

Proof. From Edmonds' characterization of the base polytope of a matroid [56], it follows that the spanning tree polytope P is defined by the following inequalities (see [121, Corollary 50.7c]):

$$\begin{aligned} P = \{\boldsymbol{z} \in \mathbb{R}^E : \ & \boldsymbol{z}(E) = |V| - 1 && && (6.6)\\ & \boldsymbol{z}(E(U)) \leq |U| - 1, && \forall U \subset V && (6.7)\\ & z_e \geq 0 && \forall e \in E.\} && (6.8) \end{aligned}$$

The relative interior of P corresponds to those $\boldsymbol{z} \in P$ satisfying all inequalities (6.7) and (6.8) *strictly*.

Clearly, $\boldsymbol{z}^*$ satisfies (6.6) since:

$$\begin{aligned} \forall v \in V,\ \boldsymbol{x}^*(\delta^+(v)) = 1 \ &\Rightarrow\ \boldsymbol{x}^*(A) = n = |V| \\ &\Rightarrow\ \boldsymbol{z}^*(E) = n - 1 = |V| - 1. \end{aligned}$$

Consider any set $U \subset V$. We have

$$\begin{aligned} \sum_{v \in U} \boldsymbol{x}^*(\delta^+(v)) \ &= \ |U| = \boldsymbol{x}^*(A(U)) + \boldsymbol{x}^*(\delta^+(U)) \\ &\geq \ x^*(A(U)) + 1. \end{aligned}$$

Since $\boldsymbol{x}^*$ satisfies (6.2) and (6.3), we have

$$\boldsymbol{z}^*(E(U)) = \frac{n-1}{n}\boldsymbol{x}^*(A(U)) < \boldsymbol{x}^*(A(U)) \leq |U| - 1,$$

showing that $\boldsymbol{z}^*$ satisfies (6.7) strictly. Since E is the support of $\boldsymbol{z}^*$, (6.8) is also satisfied strictly by $\boldsymbol{z}^*$. This shows that $\boldsymbol{z}^*$ is in the relative interior of P. □

One implication of being in the relative interior of P is that $\boldsymbol{z}^*$ can be expressed as a convex combination of spanning trees such that the coefficient corresponding to *any* spanning tree is positive.

Regarding the size of the extreme point $\boldsymbol{x}^*$, it is known that its support A has at most $3n - 4$ arcs (see [71, Theorem 15]). In addition, we know that it can be expressed as the unique solution of an invertible system with only $0 - 1$ coefficients, and therefore, we have that every entry x_a^* is rational with integral numerator and denominator bounded by $2^{O(n \ln n)}$. In particular, $z^*_{\min} = \min_{e \in E} z_e^* > 2^{-O(n \ln n)}$.

6.4 Random Spanning Tree Sampling and Concentration Bounds

Our goal in this section is to round $\boldsymbol{z}^*$, as a point in the relative interior of the spanning tree polytope, to a spanning tree. Suppose that we are looking for a distribution over

spanning trees of G that preserves the marginal probabilities imposed by $\boldsymbol{z}^*$, i.e. $\Pr[e \in T] = z_e^*$ for every edge $e \in E$, where $G = (V, E)$ is the graph whose edge set E is the support of $\boldsymbol{z}^*$. There is plenty of such distributions. The distribution we actually choose in our approach corresponds to taking the tree to be a $\boldsymbol{\lambda}$-*random tree* for some vector $\boldsymbol{\lambda} \in \mathbb{R}^{|E|}$, i.e., a spanning tree T chosen from the collection of all the possible spanning trees with probability proportional to $\prod_{e \in T} \lambda_e$ – cf. Section 2.5.2.

As we will see later – see Theorem 6.4.1 and the discussion after its statement – such a vector $\boldsymbol{\lambda}$ always exists and an approximation $\widetilde{\boldsymbol{\lambda}}$ of it (that suffices for our purposes) can be found efficiently. Furthermore, as we describe in Section 6.4.1, given $\widetilde{\boldsymbol{\lambda}}$, we can sample a $\widetilde{\boldsymbol{\lambda}}$-random tree sufficiently fast too.

The main reason behind use of such distribution is that the events corresponding to the edges of G being included in a sampled $\boldsymbol{\lambda}$-random tree are negatively correlated – see Section 6.4.2. This enables us to use the upper tail of Chernoff bound on such events. We use this tail bound – together with the union-bounding technique of Karger [86] – to establish the *thinness* of a sampled tree. (Roughly speaking, a tree is said to be thin if the number of its edges in each cut is not much higher than its expected value; see Section 6.5 for a formal definition of thinness.) This, in turn, will enable us to obtain from this sampled thin tree a tour in G that has sufficiently small cost – see Section 6.6 for details.

Before we proceed, let us just note that in Section 6.7 we show how to efficiently find $\widetilde{\lambda}_e$'s that approximately preserve the margins dictated by $\boldsymbol{z}^*$. More formally, we prove the following theorem and we use in the rest of the chapter the vector $\widetilde{\boldsymbol{\lambda}}$ that is produced by it for $\varepsilon = 0.2$.. (Recall that for our ATSP application, $z_{\min}$ is $2^{-O(n \ln n)}$ and having ε equal to 0.2 suffices.)

Theorem 6.4.1. *Given $\boldsymbol{z}$ in the spanning tree polytope of $G = (V, E)$ and some $\varepsilon > 0$, one can find values $\widetilde{\lambda}_e$, for all $e \in E$, such that if we sample a $\widetilde{\boldsymbol{\lambda}}$-random tree T then*

$$\Pr[e \in T] \leq (1 + \varepsilon) z_e,$$

for each e, i.e. the marginals are $(1 + \varepsilon)$-approximately preserved. Furthermore, the running time of the resulting algorithm is polynomial in $n = |V|$, $-\ln z_{\min}$ and $1/\varepsilon$.

The above theorem shows that for any $\boldsymbol{z}$ one can find a vector $\widetilde{\boldsymbol{\lambda}}$ that preserves marginals up to arbitrarily small precision. This suffices for our purposes. However, a natural question here is whether there always exists a vector $\boldsymbol{\lambda}$ that preserves the marginals exactly. It turns out (cf. [18]) that indeed such a vector exists, as long as, $\boldsymbol{z}$ is in the relative interior of the spanning tree polytope. Furthermore, this requirement on $\boldsymbol{z}$ is necessary (in fact, this has been observed already before, see Exercise 4.19 in [104]). To see why it is the case, suppose G is a triangle and $\boldsymbol{z}$ is the vector $(\frac{1}{2}, \frac{1}{2}, 1)$. In this case, $\boldsymbol{z}$ is in the polytope (but not in its relative interior) and there are no λ_e's that preserve the margins exactly.

6.4.1 Sampling a $\boldsymbol{\lambda}$-Random Tree

There is a host of results (cf. [78, 97, 45, 4, 33, 138, 91] and Chapter 7) on obtaining polynomial-time algorithms for generating a uniform spanning tree, i.e. a $\boldsymbol{\lambda}$-random tree for the case of all λ_es being equal. Almost all of them can be easily modified to allow arbitrary $\boldsymbol{\lambda}$; however, not all of them still guarantee a polynomial running time for such general λ_es. (For instance, the algorithm to be present in Chapter 7 might be not polynomial-time in this setting.) Therefore, to circumvent this problem we will use an iterative approach similar to [97] that is still polynomial for general λ_es.

This approach is based on ordering the edges $e_1, \ldots, e_m$ of G arbitrarily and processing them one by one, deciding probabilistically whether to add a given edge to the final tree or to discard it. More precisely, when we process the j-th edge e_j, we decide to add it to a final spanning tree T with probability p_j being the probability that e_j is in a $\boldsymbol{\lambda}$-random tree *conditioned* on the decisions that were made for edges $e_1, \ldots, e_{j-1}$ in earlier iterations. Clearly, this procedure generates a $\boldsymbol{\lambda}$-random tree, and its running time is polynomial as long as the computation of the probabilities $p_1, \ldots, p_m$ can be done in polynomial time.

To compute these probabilities efficiently we note that, by Fact 2.5.2, we have that p_1 is equal to λ_{e_1} times the effective resistance of the edge $e_1 = (u_1, v_1)$ in the graph G that has the resistance r_e of each edge e equal to $1/\lambda_e$. Furthermore, by definition of the effective resistance (cf. (2.13)), we can compute it by finding the vertex potentials corresponding to an electrical u_1-v_1 flow of value 1. These, in turn, can be obtained by solving the Laplacian system (cf. (2.10)). Note that we care only about having a polynomial running time here, so we can afford solving this system – and thus compute p_1 – exactly, e.g., by employing Gaussian elimination.

Now, once the probability p_1 has been computed, to compute p_2 we note that if we choose to include e_1 in the tree then:

$$\begin{aligned} p_2 = \Pr[e_2 \in T | e_1 \in T] &= \frac{\sum_{T' \ni e_1, e_2} \prod_{e \in T'} \lambda_e}{\sum_{T' \ni e_1} \prod_{e \in T'} \lambda_e} \\ &= \frac{\sum_{T' \ni e_1, e_2} \prod_{e \in T' \setminus e_1} \lambda_e}{\sum_{T' \ni e_1} \prod_{e \in T' \setminus e_1} \lambda_e}. \end{aligned}$$

As one can see, the probability that $e_2 \in T$ conditioned on the event that $e_1 \in T$ is equal to the probability that e_2 is in a $\boldsymbol{\lambda}$-random tree of a graph obtained from G by contracting the edge e_1. (Note that one can sum up the λ_es of the multi-edges formed during the contractions and replace them with one single edge.) Similarly, if we choose to discard e_1, the probability p_2 is equal to the probability that e_2 is in a $\boldsymbol{\lambda}$-random tree of a graph obtained from G by removing e_1. In general, p_j is equal to the probability that e_j is included in a $\boldsymbol{\lambda}$-random tree of a graph obtained from G by contracting all edges that we have already decided to add to the tree, and deleting all edges that we have already decided to discard. So, this means that we can compute each p_j in polynomial time in the manner completely analogous to the way we computed p_1 – we just use appropriately modified version of the graph G for each of these computations.

6.4.2 Negative Correlation and a Concentration Bound

We derive now the following concentration bound. As discussed in the next section, this bound is instrumental in establishing the thinness of a sampled tree.

Theorem 6.4.2. *For each edge e, let X_e be an indicator random variable associated with the event $[e \in T]$, where T is a $\boldsymbol{\lambda}$-random tree. Also, for any subset C of the edges of G, define $X(C) = \sum_{e \in C} X_e$. Then we have*

$$\Pr[X(C) \geq (1+\delta)E[X(C)]] \leq \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{E[X(C)]}.$$

Usually, when we want to obtain such concentration bounds, we prove that the variables $\{X_e\}_e$ are independent and use the Chernoff bound. Unfortunately, in our case, the variables $\{X_e\}_e$ are not independent. However, it turns out that they are still *negatively correlated*, i.e. for any subset $F \subseteq E$,

$$\Pr[\forall_{e \in F} X_e = 1] \leq \prod_{e \in F} \Pr[X_e = 1].$$

This fact should be not too surprising as, intuitively, if one considers, e.g., $F = \{e, e'\}$ then the presence of the edge e in the tree makes the edge e' "less needed" for making sure that the resulting tree is connected and more likely to lead to creation of a cycle. Also, as the number of edges in a spanning tree is exactly one fewer than the number of vertices of the underlying graph, the negative correlation holds "on average". However, despite this intuitive feel, no direct proof of this negative correlation is known, and the only proof crucially relies on the connection between random spanning trees and electrical flows, i.e., Fact 2.5.2.

Lemma 6.4.3. *The random variables $\{X_e\}_e$ are negatively correlated.*

Proof. Let us consider some subset F of the edges E and let $e_1, \ldots, e_k$ be some arbitrary ordering of the edges in F. Now, as we have

$$\Pr[\forall_{e \in F} X_e = 1] = \prod_{i=1}^{k} \Pr[e_i \in T | X_{e_j} = 1 \forall_{j<i}],$$

it just suffices to show that for any i

$$\Pr[e_i \in T | X_{e_j} = 1 \forall_{j<i}] \leq \Pr[e_i \in T].$$

To this end, we note that by Fact 2.5.2 we have that $\Pr[e_i \in T]$ is equal to λ_{e_i} times the effective resistance of e_i in a version of the graph G with resistances r_e equal to $1/\lambda_e$ for each edge e. Furthermore, by the discussion in Section 6.4.1 (and Fact 2.5.2), we know that $\Pr[e_i \in T | X_{e_j} = 1 \forall_{j<i}]$ is equal to λ_{e_i} times the effective resistance of e_i in a version of G corresponding to contracting the edges e_j for $j < i$ – which can be viewed as setting the resistances r_{e_j} of these edges to zero – and setting the resistances r_e of the rest of the edges to $1/\lambda_e$.

However, by Rayleigh's Monotonicity Principle (cf. Corollary 2.4.2), we must have that the effective resistance of e_i in the first version of the graph G can be only bigger than the effective resistance of e_i in the second version. Thus, the lemma follows. □

Once we have established negative correlation between X_es, Theorem 6.4.2 follows directly from the result of Panconesi and Srinivasan [111] that the upper tail part of the Chernoff bound requires only negative correlation (or even a weaker notion, see [111]) and not the full independence of the random variables.

At this point, we want to note that other ways of producing *negatively correlated* probability distributions on trees (or, more generally, matroid bases) satisfying some given marginals $\boldsymbol{z}$ have been proposed by Chekuri, Vondrak and Zenklusen [39]. Their approach can also be used in the framework developed in this chapter.

6.5 The Thinness Property

In this section, we focus on the $\widetilde{\boldsymbol{\lambda}}$-random trees that we obtain by applying the algorithm of Theorem 6.4.1 to $\boldsymbol{z}^*$. We show that such trees are almost surely "thin".

We start by defining the thinness property formally.

Definition 6.5.1. *We say that a tree T is α-thin if for each set $U \subset V$,*

$$|T \cap \delta(U)| \leq \alpha \boldsymbol{z}^*(\delta(U)).$$

Also we say that T is (α, s)-thin if it is α-thin and moreover,

$$c(T) \leq s OPT_{HK}.$$

We first prove that if we focus on a particular cut then the "α-thinness" property holds for it with overwhelming probability where $\alpha \approx \frac{\log n}{\log \log n}$.

Lemma 6.5.2. *If T is a $\widetilde{\boldsymbol{\lambda}}$-random tree for $\widetilde{\boldsymbol{\lambda}}$ corresponding to applying Theorem 6.4.1 to $\boldsymbol{z}^*$ with $\varepsilon = 0.2$ in a graph G with $n \geq 5$ vertices, then for any set $U \subset V$,*

$$\Pr[|T \cap \delta(U)| > \beta \boldsymbol{z}^*(\delta(U))] \leq n^{-2.5\boldsymbol{z}^*(\delta(U))},$$

where $\beta = 4 \ln n / \ln \ln n$.

Proof. Note that by definition, for all edges $e \in E$, $\Pr[e \in T] \leq (1+\varepsilon) z^*_e$, where $\varepsilon = 0.2$ is our desired accuracy parameter. Hence,

$$\mathrm{E}[|T \cap \delta(U)|] = \sum_{e \in \delta(U))} \Pr[e \in T] = \tilde{z}\left(\delta(U)\right) \leq (1+\varepsilon) \boldsymbol{z}^*(\delta(U)),$$

where we define $\tilde{z}\left(\delta(U)\right) := \sum_{e \in \delta(U))} \Pr[e \in T]$. Applying Theorem 6.4.2 with

$$1 + \delta = \frac{\beta \boldsymbol{z}^*(\delta(U))}{\tilde{z}\left(\delta(U)\right)} \geq \frac{\beta}{1+\varepsilon},$$

we derive that $\Pr[|T \cap \delta(U)| > \beta \boldsymbol{z}^*(\delta(U))]$ can be bounded from above by

$$\begin{aligned}
&\Pr[|T \cap \delta(U)| > (1+\delta)\mathrm{E}[|T \cap \delta(U)|]] \\
\leq\ & \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^{\tilde{z}(\delta(U))} \\
\leq\ & \left(\frac{e}{1+\delta}\right)^{(1+\delta)\tilde{z}(\delta(U))} \\
=\ & \left(\frac{e}{1+\delta}\right)^{\beta \boldsymbol{z}^*(\delta(U))} \\
\leq\ & \left[\left(\frac{e(1+\varepsilon)}{\beta}\right)^{\beta}\right]^{\boldsymbol{z}^*(\delta(U))} \\
\leq\ & n^{-4(1-1/e)\boldsymbol{z}^*(\delta(U))},
\end{aligned}$$

where, in the last inequality, we have used that

$$\begin{aligned}
\ln\left[\left(\frac{e(1+\varepsilon)}{\beta}\right)^{\beta}\right] &= 4\frac{\ln n}{\ln\ln n}[1+\ln(1+\varepsilon)-\ln(4) \\
&\qquad -\ln\ln n+\ln\ln\ln n] \\
&\leq -4\ln n\left(1-\frac{\ln\ln\ln n}{\ln\ln n}\right) \\
&\leq -4\left(1-\frac{1}{e}\right)\ln n \leq -2.5\ln n,
\end{aligned}$$

since $e(1+\varepsilon) < 4$ and $\frac{\ln\ln\ln n}{\ln\ln n} \leq \frac{1}{e}$ for all $n \geq 5$ (even for $n \geq 3$). □

We are ready to prove the main theorem of this section.

Theorem 6.5.3. *Let $n \geq 5$ and $\varepsilon = 0.2$. Let $T_1, \ldots, T_{\lceil 2\ln n\rceil}$ be $\lceil 2\ln n\rceil$ independent sampled $\widetilde{\boldsymbol{\lambda}}$-random trees where $\widetilde{\lambda}_e$s are as given in Theorem 6.4.1. Let T^* be the tree among these samples that minimizes $c(T_j)$. Then, T^* is $(4\ln n/\ln\ln n, 2)$-thin with high probability.*

Here, high probability means probability at least $1-1/n$, but probability $1-1/n^k$ can be achieved by sampling $2k\ln n$ trees.

Proof. We start by showing that for any $1 \leq j \leq \lceil 2\ln n\rceil$, T_j is β-thin with high probability for $\beta = 4\ln n/\ln\ln n$. From Lemma 6.5.2 we know that the probability of some particular cut $\delta(U)$ violating the β-thinness of T_j is at most $n^{-2.5\boldsymbol{z}^*(\delta(U))}$. Now, we use a result of Karger [86] that shows that there are at most n^{2l} cuts of size at most l times the minimum cut value for any half-integer $l \geq 1$. Since, by the definitions of the Held-Karp relaxation and of $\boldsymbol{z}^*$, we know that $\boldsymbol{z}^*(\delta(U)) \geq 2(1-1/n)$, it means there is at most n^l cuts $\delta(U)$ with $\boldsymbol{z}^*(\delta(U)) \leq l(1-1/n)$ for any integer $l \geq 2$.

Therefore, by applying the union bound (and $n \geq 5$), we derive that the probability that there exists some cut $\delta(U)$ with $|T_j \cap \delta(U)| > \beta \boldsymbol{z}^*(\delta(U))$ is at most

$$\sum_{i=3}^{\infty} n^i n^{-2.5(i-1)(1-1/n)},$$

where each term is an upper bound on the probability that there exists a violating cut of size within $[(i-1)(1-1/n), i(1-1/n)]$. Fo $n \geq 5$, this simplifies to:

$$\sum_{i=3}^{\infty} n^i n^{-2.5(i-1)(1-1/n)} \leq \sum_{i=3}^{\infty} n^{-i+2} = \frac{1}{n-1},$$

Thus, indeed, T_j is a β-thin spanning tree with high probability.

Now, the expected cost of T_j is

$$\mathrm{E}[c(T_j)] \leq \sum_{e \in E} \tilde{z}_e \leq (1+\varepsilon)\frac{n-1}{n} \sum_{a \in A} x^*_a \leq (1+\varepsilon)\mathrm{OPT}_{\mathrm{HK}}.$$

So, by Markov inequality we have that for any j, the probability that $c(T_j) > 2\mathrm{OPT}_{\mathrm{HK}}$ is at most $(1+\varepsilon)/2$. Thus, with probability at most $(\frac{1+\varepsilon}{2})^{2\ln n} < \frac{1}{n}$ for $\varepsilon = 0.2$, we have $c(T^*) > 2\mathrm{OPT}_{\mathrm{HK}}$. This concludes the proof of the theorem. □

6.6 Transforming a Thin Spanning Tree into an Eulerian Walk

As the final step of the algorithm, we show how one can find an Eulerian walk with small cost using a thin tree. After finding such a walk, one can use the metric property to convert this walk into an Hamiltonian cycle of no greater cost (by shortcutting). In particular, the following theorem justifies the definition of thin spanning trees.

Theorem 6.6.1. *Assume that we are given an (α, s)-thin spanning tree T^* with respect to the LP relaxation $\boldsymbol{x}^*$. Then we can find a Hamiltonian cycle of cost no more than $(2\alpha + s)c(\boldsymbol{x}^*) = (2\alpha + s)OPT_{HK}$ in polynomial time.*

Before proceeding to the proof of Theorem 6.6.1, we recall some basic network flow results related to circulations. A function $f : A \to \mathbb{R}$ is called a *circulation* if $f(\delta^+(v)) = f(\delta^-(v))$ for each vertex $v \in V$. Hoffman's circulation theorem [121, Theorem 11.2] gives a necessary and sufficient condition for the existence of a circulation subject to lower and upper capacities on arcs.

Theorem 6.6.2 (Hoffman's circulation theorem)**.** *Given lower and upper capacities $l, u : A \to \mathbb{R}$, there exists a circulation f satisfying $l(a) \leq f(a) \leq u(a)$ for all $a \in A$ if and only if*

1. *$l(a) \leq u(a)$ for all $a \in A$ and*

2. for all subsets $U \subset V$, we have $l(\delta^-(U)) \leq u(\delta^+(U))$.

Furthermore, if l and u are integer-valued, f can be chosen to be integer-valued.

Proof of Theorem 6.6.1. We first orient each edge $\{u, v\}$ of T^* to $\arg\min\{c(a) : a \in \{(u, v), (v, u)\} \cap A\}$, and denote the resulting directed tree by $\vec{T}^*$. Observe that by definition of our undirected cost function, we have $c(\vec{T}^*) = c(T^*)$. We then find a minimum cost augmentation of $\vec{T}^*$ into an Eulerian directed graph; this can be formulated as a minimum cost circulation problem with integral lower capacities (and no or infinite upper capacities). Indeed, set

$$l(a) = \begin{cases} 1 & a \in \vec{T}^* \\ 0 & a \notin \vec{T}^*, \end{cases}$$

and consider the minimum cost circulation problem

$$\min\{c(f) : f \text{ is a circulation and } f(a) \geq l(a)\ \forall a \in A\}.$$

An optimum circulation f^* can be computed in polynomial time and can be assumed to be integral, see e.g. [121, Corollary 12.2a]. This integral circulation f^* corresponds to a directed (multi)graph H which contains $\vec{T}^*$. Every vertex in H has an indegree equal to its outdegree. Therefore, every cut has the same number of arcs in both directions. As H is weakly connected (as it contains $\vec{T}^*$), it is strongly connected and thus, H is an Eulerian directed multigraph. We can extract an Eulerian walk of H and shortcut it to obtain our Hamiltonian cycle of cost at most $c(f^*)$ since the costs satisfy the triangle inequality.

To complete the proof of Theorem 6.6.1, it remains to show that $c(f^*) \leq (2\alpha + s)c(\boldsymbol{x}^*)$. For this purpose, we define

$$u(a) = \begin{cases} 1 + 2\alpha x^*_a & a \in \vec{T}^* \\ 2\alpha x^*_a & a \notin \vec{T}^*. \end{cases}$$

We claim that there exists a circulation g satisfying $l(a) \leq g(a) \leq u(a)$ for every $a \in A$. To prove this claim, we use Hoffman's circulation theorem 6.6.2. Indeed, by construction, $l(a) \leq u(a)$ for every $a \in A$; furthermore, Lemma 6.6.3 below shows that, for every $U \subset V$, we have $l(\delta^-(U)) \leq u(\delta^+(U))$. Thus the existence of the circulation g is established. Furthermore,

$$c(f^*) \leq c(g) \leq c(u) = c(\vec{T}^*) + 2\alpha c(\boldsymbol{x}^*) \leq (2\alpha + s)c(\boldsymbol{x}^*),$$

establishing the bound on the cost of f^*. This completes the proof of Theorem 6.6.1. □

Lemma 6.6.3. *For the capacities l and u as constructed in the proof of Theorem 6.6.1, the following holds for any subset $U \subset V$:*

$$l(\delta^-(U)) \leq u(\delta^+(U)).$$

Proof. Irrespective of the orientation of T^* into $\vec{T}^*$, the number of arcs of $\vec{T}^*$ in $\delta^-(U)$ is at most $\alpha \boldsymbol{z}^*(\delta(U))$ by definition of α-thinness. Thus

$$l(\delta^-(U)) \leq \alpha \boldsymbol{z}^*(\delta(U)) < 2\alpha \boldsymbol{x}^*(\delta^-(U)),$$

due to (6.4) and (6.5). On the other hand, we have

$$u(\delta^+(U)) \geq 2\alpha \boldsymbol{x}^*(\delta^+(U)) = 2\alpha \boldsymbol{x}^*(\delta^-(U)) \geq l(\delta^-(U)),$$

where we have used the fact that $\boldsymbol{x}^*$ itself is a circulation (see (6.4)). The lemma follows. □

Theorem 6.6.4. *For a suitable choice of parameters, the algorithm given in Figure 6-1 finds a* $(2+8\ln n/\ln\ln n)$*-approximate solution to the asymmetric traveling salesman problem with high probability and in time polynomial in the size of the input.*

Proof. The algorithm start by finding an optimal extreme-point solution $\boldsymbol{x}^*$ to the Held-Karp LP relaxation of ATSP of value $\mathrm{OPT}_{\mathrm{HK}}$. Next, using the algorithm of Theorem 6.4.1 on $\boldsymbol{z}^*$ (which is defined by (6.5)) with $\varepsilon = 0.2$, we obtain $\widetilde{\lambda}_e$s. Since $\boldsymbol{x}^*$ was an extreme point, we know that $z^*_{\min} \geq e^{-O(n \ln n)}$; thus, the algorithm of Theorem 6.4.1 indeed runs in polynomial time.

Next, we use the polynomial time sampling procedure described in section 6.4.1 to sample $\Theta(\ln n)$ $\widetilde{\boldsymbol{\lambda}}$-random trees T_j, and take T^* to be the one among them that minimizes the cost $c(T_j)$. By Theorem 6.5.3, we know that T^* is $(\beta, 2)$-thin with high probability.

Now, we use Theorem 6.6.1 to obtain, in polynomial time, a $(2 + 8\ln n/\ln\ln n)$-approximation of our ATSP instance. □

The proof also shows that the integrality gap of the Held-Karp relaxation for the asymmetric TSP is bounded above by $2+8\ln n/\ln\ln n$. The best known lower bound on the integrality gap is only 2, as shown in [36]. Closing this gap is a challenging open question, and this possibly could be answered using thinner spanning trees. In particular, we formulate the following conjecture.

Corollary 6.6.5. *There always exists a* (C_1, C_2)*-thin spanning tree where* C_1 *and* C_2 *are constants. Therefore, the integrality gap of the ATSP Held-Karp linear programming relaxation* (6.1) *is a constant.*

6.7 A Combinatorial Algorithm for Finding $\widetilde{\boldsymbol{\lambda}}$s

In this section, we provide a combinatorial algorithm to efficiently find $\widetilde{\lambda}_e$s such that the $\widetilde{\boldsymbol{\lambda}}$-random tree approximately preserves the marginal probabilities given by a point $\boldsymbol{z}$ in the spanning tree polytope and therefore prove Theorem 6.4.1. To this end, let us fix some accuracy parameter $\varepsilon > 0$ and, for presentation reasons, let us focus on finding $\widetilde{\gamma}$ such that $\widetilde{\boldsymbol{\lambda}} := e^{\widetilde{\gamma}}$ will have the desired properties.

Given a vector γ, for each edge e, define $q_e(\gamma) := \frac{\sum_{T \ni e} \exp(\gamma(T))}{\sum_T \exp(\gamma(T))}$, where $\gamma(T) = \sum_{f \in T} \gamma_f$. For notational convenience, we have dropped the fact that $T \in \mathcal{T}$ in these summations; this shouldn't lead to any confusion. Restated, $q_e(\gamma)$ is the probability that edge e will be included in a spanning tree T that is chosen with probability proportional to $\exp(\gamma(T))$.

We compute $\widetilde{\gamma}$ using the following simple algorithm. Start with all γ_e equal, and as long as the marginal $q_e(\gamma)$ for some edge e is more than $(1+\varepsilon)z_e$, we decrease appropriately γ_e in order to decrease $q_e(\gamma)$ to $(1+\varepsilon/2)z_e$. More formally, here is a description of the algorithm.

1. Set $\gamma = \vec{0}$.

2. While there exists an edge e with $q_e(\gamma) > (1+\varepsilon)z_e$:

   - Compute δ such that if we define γ' as $\gamma'_e = \gamma_e - \delta$, and $\gamma'_f = \gamma_f$ for all $f \in E \setminus \{e\}$, then $q_e(\gamma') = (1+\varepsilon/2)z_e$
   - Set $\gamma \leftarrow \gamma'$

3. Output $\widetilde{\gamma} := \gamma$.

Clearly, *if* the above procedure terminates then the resulting $\widetilde{\gamma}$ satisfies the requirement of Theorem 6.4.1. Therefore, what we need to show is that this algorithm terminates in time polynomial in n, $-\ln z_{\min}$ and $1/\varepsilon$, and that each iteration can be implemented in polynomial time.

We start by bounding the number of iterations - we will show that it is

$$O(\frac{1}{\varepsilon}|E|^2[|V|\ln(|V|) - \ln(\varepsilon z_{\min})]).$$

In the next lemma, we derive an equation for δ, and prove that for $f \neq e$ the probabilities $q_f(\cdot)$ do not decrease as a result of decreasing γ_e.

Lemma 6.7.1. *If for some $\delta \geq 0$ and an edge e, we define γ' by $\gamma'_e = \gamma_e - \delta$ and $\gamma'_f = \gamma_f$ for all $f \neq e$, then*

1. *for all $f \in E \setminus \{e\}$, $q_f(\gamma') \geq q_f(\gamma)$,*

2. *$q_e(\gamma')$ satisfies $\frac{1}{q_e(\gamma')} - 1 = e^{\delta}\left(\frac{1}{q_e(\gamma)} - 1\right)$.*

In particular, in the main loop of the algorithm, since $q_e(\gamma) > (1+\varepsilon)z_e$ and we want $q_e(\gamma') = (1+\varepsilon/2)z_e$, we get $\delta = \ln \frac{q_e(\gamma)(1-(1+\varepsilon/2)z_e)}{(1-q_e(\gamma))(1+\varepsilon/2)z_e} > \ln \frac{(1+\varepsilon)}{(1+\varepsilon/2)} > \frac{\varepsilon}{4}$ for $\varepsilon \leq 1$ (for larger values of ε, we can simply decrease ε to 1).

Proof. Let us consider some $f \in E \setminus \{e\}$. We have

$$\begin{aligned} q_f(\gamma') &= \frac{\sum_{T \in \mathcal{T}: f \in T} \exp(\gamma'(T))}{\sum_{T \in \mathcal{T}} \exp(\gamma'(T))} \\ &= \frac{\sum_{T: e \in T, f \in T} e^{\gamma'(T)} + \sum_{T: e \notin T, f \in T} e^{\gamma'(T)}}{\sum_{T \ni e} e^{\gamma'(T)} + \sum_{T: e \notin T} e^{\gamma'(T)}} \\ &= \frac{e^{-\delta} \sum_{T: e \in T, f \in T} e^{\gamma(T)} + \sum_{T: e \notin T, f \in T} e^{\gamma(T)}}{e^{-\delta} \sum_{T \ni e} e^{\gamma(T)} + \sum_{T: e \notin T} e^{\gamma(T)}} \\ &= \frac{e^{-\delta} a + b}{e^{-\delta} c + d} \end{aligned}$$

with a, b, c, d appropriately defined. The same expression holds for $q_f(\gamma)$ with the $e^{-\delta}$ factors removed. But, for general $a, b, c, d \geq 0$, if $\frac{a}{c} \leq \frac{a+b}{c+d}$ then $\frac{xa+b}{xc+d} \geq \frac{a+b}{c+d}$ for $x \leq 1$. Since

$$\frac{a}{c} = \frac{\sum_{T \in \mathcal{T}: e \in T, f \in T} e^{\gamma(T)}}{\sum_{T \in \mathcal{T}: e \in T} e^{\gamma(T)}} \leq q_f(\gamma) = \frac{a+b}{c+d}$$

by negative correlation (since a/c represents the conditional probability that f is present given that e is present), we get that $q_f(\gamma') \geq q_f(\gamma)$ for $\delta \geq 0$.

Now, we proceed to deriving the equation for δ. By definition of $q_e(\gamma)$, we have

$$\frac{1}{q_e(\gamma)} - 1 = \frac{\sum_{T: e \notin T} e^{\gamma(T)}}{\sum_{T \ni e} e^{\gamma(T)}}.$$

Hence,

$$\begin{aligned} \frac{1}{q_e(\gamma')} - 1 &= \frac{\sum_{T: e \notin T} e^{\gamma'(T)}}{\sum_{T \ni e} e^{\gamma'(T)}} \\ &= \frac{\sum_{T: e \notin T} e^{\gamma(T)}}{e^{-\delta} \sum_{T \ni e} e^{\gamma(T)}} \\ &= e^{\delta} \left(\frac{1}{q_e(\gamma)} - 1 \right). \end{aligned}$$

□

Before bounding the number of iterations, we collect some basic results regarding spanning trees which we need for the proof of the number of iterations.

Lemma 6.7.2. *Let $G = (V, E)$ be a graph with weights γ_e for $e \in E$. Let $Q \subset E$ be such that for all $f \in Q$, $e \in E \setminus Q$, we have $\gamma_f > \gamma_e + \Delta$ for some $\Delta \geq 0$. Let r be the size of a maximum spanning forest of Q. Then*

1. *For any $T \in \mathcal{T}$, we have $|T \cap Q| \leq r$.*

   Define $\mathcal{T}_= := \{T \in \mathcal{T} : |T \cap Q| = r\}$ and $\mathcal{T}_< := \{T \in \mathcal{T} : |T \cap Q| < r\}$.

2. *Any spanning tree $T \in \mathcal{T}_=$ can be generated by taking the union of any spanning forest F (of cardinality r) of the graph (V, Q) and a spanning tree (of cardinality $n - r - 1$) of the graph G/Q in which the edges of Q have been contracted.*

3. *Let $T_{\max}$ be a maximum spanning tree of G with respect to the weights $\gamma(\cdot)$, i.e. $T_{\max} = \arg\max_{T \in \mathcal{T}} \gamma(T)$. Then, for any $T \in \mathcal{T}_<$, we have $\gamma(T) < \gamma(T_{\max}) - \Delta$.*

Proof. These properties easily follow from the matroidal properties of spanning trees. To prove 3., consider any $T \in \mathcal{T}_<$. Since $|T \cap Q| < r$, there exists an edge $f \in (T_{\max} \cap Q) \setminus T$ such that $(T \cap Q) \cup \{f\}$ is a forest of G. Therefore, the unique circuit in $T \cup \{f\}$ contains an edge $e \notin Q$. Thus $T' = T \cup \{f\} \setminus \{e\}$ is a spanning tree. Our assumption on Q implies that

$$\gamma(T_{\max}) \geq \gamma(T') = \gamma(T) - \gamma_e + \gamma_f > \gamma(T) + \Delta,$$

which yields the desired inequality. □

We proceed to bounding the number of iterations.

Lemma 6.7.3. *The algorithm executes at most $O(\frac{1}{\varepsilon}|E|^2[|V|\ln(|V|) - \ln(\varepsilon z_{\min})])$ iterations of the main loop.*

Proof. Let $n = |V|$ and $m = |E|$. Assume for the sake of contradiction that the algorithm executes more than

$$\tau := \frac{4}{\varepsilon} m^2 [n \ln n - \ln(\varepsilon z_{\min})]$$

iterations. Let γ be the vector of γ_e's computed at such an iteration. For brevity, let us define $q_e := q_e(\gamma)$ for all edges e.

We prove first that there exists some $e^* \in E$ such that $\gamma_{e^*} < -\frac{\varepsilon\tau}{4m}$. Indeed, there are m edges, and by Lemma 6.7.1 we know that in each iteration we decrease γ_e of one of these edges by at least $\varepsilon/4$. Thus, we know that, after more than τ iterations, there exists e^* for which γ_{e^*} is as desired.

Note that we never decrease γ_e for edges e with $q_e(\cdot)$ smaller than $(1+\varepsilon)z_e$, and Lemma 6.7.1 shows that reducing γ_f of edge $f \neq e$ can only increase $q_e(\cdot)$. Therefore, we know that all the edges with γ_e being negative must satisfy $q_e \geq (1+\varepsilon/2)z_e$. In other words, all edges e such that $q_e < (1+\varepsilon/2)z_e$ satisfy $\gamma_e = 0$. Finally, by a simple averaging argument, we know that $\sum_e q_e = n-1 < (1+\varepsilon/2)(n-1) = (1+\varepsilon/2)\sum_e z_e$. Hence, there exists at least one edge f^* with $q_{f^*} < (1+\varepsilon/2)z_{f^*}$ and thus having $\gamma_{f^*} = 0$.

We proceed now to exhibiting a set Q such that:

(I)**:** $\emptyset \neq Q \subset E$, and

(II)**:** for all $e \in E \setminus Q$ and $f \in Q$, $\gamma_e + \frac{\varepsilon\tau}{4m^2} < \gamma_f$.

We construct Q as follows. We set threshold values $\Gamma_i = -\frac{\varepsilon\tau i}{4m^2}$, for $i \geq 0$, and define $Q_i = \{e \in E \,|\gamma_e \geq \Gamma_i\}$. Let $Q = Q_j$ where j is the first index such that $Q_j = Q_{j+1}$.

Clearly, by construction of Q, property (II) is satisfied. Also, Q is non-empty since $f^* \in Q_0 \subseteq Q_j = Q$. Finally, by the pigeonhole principle, since we have m different edges, we know that $j < m$. Thus, for each $e \in Q$ we have $\gamma_e > \Gamma_m = -\frac{\varepsilon\tau}{4m}$. This means that $e^* \notin Q$ and thus Q has property (I).

Observe that Q satisfies the hypothesis of Lemma 6.7.2 with $\Delta = \frac{\varepsilon\tau}{4m^2}$. Thus, for any $T \in \mathcal{T}_<$, we have

$$\gamma(T_{\max}) > \gamma(T) + \frac{\varepsilon\tau}{4m^2}, \tag{6.9}$$

where $T_{\max}$ and r are as defined in Lemma 6.7.2.

Let $\widehat{G}$ be the graph G/Q obtained by contracting all the edges in Q. So, $\widehat{G}$ consists only of edges not in Q (some of them can be self-loops). Let $\hat{\mathcal{T}}$ be the set of all spanning trees of $\widehat{G}$, and for any given edge $e \notin Q$, let $\hat{q}_e := \frac{\sum_{\widehat{T}\in\hat{\mathcal{T}},\widehat{T}\ni e}\exp(\gamma(\widehat{T}))}{\sum_{\widehat{T}\in\hat{\mathcal{T}}}\exp(\gamma(\widehat{T}))}$ be the probability that edge e is included in a random spanning tree $\widehat{T}$ of $\widehat{G}$, where each tree $\widehat{T}$ is chosen with probability proportional to $e^{\gamma(\widehat{T})}$. Since spanning trees of $\widehat{G}$ have $n-r-1$ edges, we have

$$\sum_{e\in E\setminus Q} \hat{q}_e = n - r - 1. \tag{6.10}$$

On the other hand, since z satisfies $z(E) = n-1$ and $z(Q) \leq r$ (by definition of r, see Lemma 6.7.2, part 1.), we have that $z(E \setminus Q) \geq n - r - 1$. Therefore, (6.10) implies that there must exist $\hat{e} \notin Q$ such that $\hat{q}_{\hat{e}} \leq z_{\hat{e}}$.

Our final step is to show that for any $e \notin Q$, $q_e < \hat{q}_e + \frac{\varepsilon z_{\min}}{2}$. Note that once we establish this, we know that $q_{\hat{e}} < \hat{q}_{\hat{e}} + \frac{\varepsilon z_{\min}}{2} \leq (1 + \frac{\varepsilon}{2}) z_{\hat{e}}$, and thus it must be the case that $\gamma_{\hat{e}} = 0$. But this contradicts the fact that $\hat{e} \notin Q$, as by construction all e with $\gamma_e = 0$ must be in Q. Thus, we obtain a contradiction that concludes the proof of the Lemma.

It remains to prove that for any $e \notin Q$, $q_e < \hat{q}_e + \frac{\varepsilon z_{\min}}{2}$. We have that

$$\begin{aligned}
q_e &= \frac{\sum_{T\in\mathcal{T}:e\in T} e^{\gamma(T)}}{\sum_{T\in\mathcal{T}} e^{\gamma(T)}} \\
&= \frac{\sum_{T\in\mathcal{T}_=:e\in T} e^{\gamma(T)} + \sum_{T\in\mathcal{T}_<:e\in T} e^{\gamma(T)}}{\sum_{T\in\mathcal{T}} e^{\gamma(T)}} \\
&\leq \frac{\sum_{T\in\mathcal{T}_=:e\in T} e^{\gamma(T)}}{\sum_{T\in\mathcal{T}_=} e^{\gamma(T)}} + \frac{\sum_{T\in\mathcal{T}_<:e\in T} e^{\gamma(T)}}{\sum_{T\in\mathcal{T}} e^{\gamma(T)}} \\
&\leq \frac{\sum_{T\in\mathcal{T}_=:e\in T} e^{\gamma(T)}}{\sum_{T\in\mathcal{T}_=} e^{\gamma(T)}} + \sum_{T\in\mathcal{T}_<:e\in T} \frac{e^{\gamma(T)}}{e^{\gamma(T_{\max})}},
\end{aligned} \tag{6.11}$$

the first inequality following from replacing $\mathcal{T}$ with $\mathcal{T}_=$ in the first denominator, and the second inequality following from considering only one term in the second denominator. Using (6.9) and the fact that the number of spanning trees is at most

n^{n-2}, the second term is bounded by:

$$\begin{aligned}\sum_{T\in\mathcal{T}_<:e\in T}\frac{e^{\gamma(T)}}{e^{\gamma(T_{\max})}} &\leq n^{n-2}e^{-\varepsilon\tau/4m^2} \\ &< \frac{1}{2}n^n e^{-\varepsilon\tau/4m^2} \\ &= \frac{\varepsilon z_{\min}}{2},\end{aligned} \tag{6.12}$$

by definition of τ. To handle the first term of (6.11), we can use part 2. of Lemma 6.7.2 and factorize:

$$\sum_{T\in\mathcal{T}_=} e^{\gamma(T)} = \left(\sum_{\widehat{T}\in\hat{\mathcal{T}}} e^{\gamma(\widehat{T})}\right)\left(\sum_{T'\in\mathcal{F}} e^{\gamma(T')}\right),$$

where $\mathcal{F}$ is the set of all spanning forests of (V,Q). Similarly, we can write

$$\sum_{T\in\mathcal{T}_=,T\ni e} e^{\gamma(T)} = \left(\sum_{\widehat{T}\in\hat{\mathcal{T}},\widehat{T}\ni e} e^{\gamma(\widehat{T})}\right)\left(\sum_{T'\in\mathcal{F}} e^{\gamma(T')}\right).$$

As a result, we have that the first term of (6.11) reduces to:

$$\frac{\left(\sum_{\widehat{T}\in\hat{\mathcal{T}}} e^{\gamma(\widehat{T})}\right)\left(\sum_{T'\in\mathcal{F}} e^{\gamma(T')}\right)}{\left(\sum_{\widehat{T}\in\hat{\mathcal{T}},\widehat{T}\ni e} e^{\gamma(\widehat{T})}\right)\left(\sum_{T'\in\mathcal{F}} e^{\gamma(T')}\right)} = \frac{\sum_{\widehat{T}\in\hat{\mathcal{T}}} e^{\gamma(\widehat{T})}}{\sum_{\widehat{T}\in\hat{\mathcal{T}},\widehat{T}\ni e} e^{\gamma(\widehat{T})}} = \hat{q}_e.$$

Together with (6.11) and (6.12), this gives

$$q_e \leq \hat{q}_e + \frac{\varepsilon z_{\min}}{2},$$

which completes the proof. □

To complete the analysis of the algorithm, we need to argue that each iteration can be implemented in polynomial time. First, for any given vector γ, we can compute efficiently the sums $\sum_T \exp(\gamma(T))$ and $\sum_{T\ni e}\exp(\gamma(T))$ for any edge e - this will enable us to compute all $q_e(\gamma)$'s. As $\sum_T \exp(\gamma(T)) = \sum_T \prod_{e\in T}\lambda_e$ and $\sum_{T\ni e}\exp(\gamma(T)) = \sum_{T\ni e}\prod_{e\in T}\lambda_e$, we can compute these terms by using Fact 2.5.2 and Fact 2.5.3. Observe that we can bound all entries of the weighted Laplacian matrix we are dealing with here, in terms of the input size, since the proof of Lemma 6.7.3 actually shows that $-\frac{\varepsilon\tau}{4|E|} \leq \gamma_e \leq 0$ for all $e\in E$ and any iteration of the algorithm. Therefore, we can compute these terms, in time polynomial in n, $-\ln z_{\min}$ and $1/\varepsilon$, as desired. Finally, δ can be computed efficiently from Lemma 6.7.1.

Chapter 7

Generation of an Uniformly Random Spanning Tree

In this chapter, we set forth a new algorithm for generating uniformly random spanning trees in undirected graphs.[1] Our algorithm produces a sample from the uniform distribution over all the spanning trees of the underlying graph in expected time $\widetilde{O}(m\sqrt{n})$. This improves the sparse graph case of the best previously known worst-case bound of $O(\min\{mn, n^{2.376}\})$.

To achieve this result, we exploit the connection between random walks and electrical flows (cf. Section 2.5.2) to integrate the traditional random-walk-based techniques for this problem with combinatorial graph partitioning primitives and fast Laplacian system solvers discussed in Section 2.7.

7.1 Introduction

The topic of this chapter is generation of uniformly random spanning trees in undirected graphs. Random spanning trees are among the oldest and most extensively investigated probabilistic objects in graph theory, with their study dating back to Kirchhoff's work in the 1840s [93]. However, it is only in the past several decades that researchers have taken up the question of how best to generate uniformly random spanning trees algorithmically. This question became an active area of research in the 1980s and 1990s, during which a long string of papers appeared that provided successively faster algorithms for this task (e.g.,[78, 97, 44, 45, 33, 4, 83, 138]).

Previous algorithms for this problem broadly fall into two categories: determinant-based algorithms and random-walk-based algorithms. The starting point for the determinant-based algorithms was Kirchhoff's Matrix Tree Theorem, which reduces counting the number of spanning trees in a graph to the evaluation of a determinant [93] (see, e.g., Ch.2, Thm. 8 in [31]). The first such algorithm produced a random spanning tree in time $O(mn^3)$ [78, 97]. After sequence of improvements ([44, 43]), this line of research culminated in the algorithm of Colbourn, Myrvold, Day, and Nel

[1]This chapter is based on joint work with Jonathan Kelner and James Propp, and contains material from [91].

[45], which runs in the amount of time necessary to multiply two $n \times n$ matrices, the best known bound for which is $O(n^{2.376})$ [46].

The random-walk-based algorithms began with the following striking theorem due to Broder [33] and Aldous [4]

Theorem 7.1.1. *Suppose you simulate a random walk in an undirected graph $G = (V, E)$, starting from an arbitrary vertex s and continuing until every vertex has been visited. For each vertex $v \in V \setminus \{s\}$, let e_v be the edge through which v was visited for the first time in this walk. Then, $T = \{e_v \mid v \in V \setminus \{s\}\}$ is a uniformly random spanning tree of G.*

This immediately yields an algorithm for generating a random spanning tree whose running time is proportional to the cover time of G. If G has n vertices and m edges, the cover time can be $\Theta(mn)$ in the worst case, but it is often much smaller. For sufficiently sparse graphs, this yields a better worst-case running time than the determinant-based algorithms. Now, since one clearly needs to see every vertex in G, it would seem unlikely that such methods could run in less than the cover time of the graph. However, in the last major breakthrough in this line of research, Wilson [138] showed that, by using a different random process, one could actually generate spanning trees in expected time proportional to the mean hitting time of the graph, which can be much smaller than the cover time (but has the same worst-case asymptotics).

The worst-case running time bound of $O(mn)$ has stood for twenty years. In this chapter, our main result is a new algorithm that offers a better worst-case time bound:

Theorem 7.1.2. *Let G be a graph with n vertices and m edges. We can generate an uniformly random spanning tree of G in expected time $\widetilde{O}(m\sqrt{n})$.*

Beyond the classical applications of generating random spanning trees that motivated the original work on the problem, there have been some developments that further motivate their study. In particular, Goyal, Rademacher, and Vempala [75] showed how to use random spanning trees to generate efficient sparsifiers of a graph, and they then explained how this could be used to provide a scalable and robust routing scheme.

7.1.1 Random Spanning Trees and Arborescences

Formally, we consider the following problem: given an undirected graph $G = (V, E)$ with n vertices and m edges, find a randomized algorithm A that, for each spanning tree T of G, outputs T with probability $1/|\mathcal{T}(G)|$, where $\mathcal{T}(G)$ is the set of all spanning trees of G.

For technical reasons that will arise later, it will be useful to consider arborescences in addition to trees. For a given $s \in G$, an *arborescence T rooted at s* is a directed spanning tree of G in which all vertices in $G \setminus \{s\}$ have exactly one incoming arc. We use the notation $r(T)$ to denote the root of an arborescence, and we use $e_T(v)$, for any $v \in G \setminus \{r(T)\}$, to denote the unique arc incoming to v in T.

We say that a procedure A *generates a conditionally random arborescence* if it outputs a vertex s and an arborescence T rooted at s such that the probability, conditioned on some s being a root, of outputting a particular arborescence T rooted at s is $1/|\mathcal{T}_s(G)|$, where $\mathcal{T}_s(G)$ is the set of all arborescences in G rooted at s.[2] Note that in this definition we do not make any restrictions on the probabilities $p_A(s)$ that the generated arborescence output by A is rooted at s. Now, it is easy to see that, once we fix some $s \in G$, there is one-to-one correspondence between spanning trees of G and arborescences rooted at s. Indeed, given any spanning tree, there is a unique way of directing its edges to make it a valid arborescence rooted at s; conversely, given any arborescence rooted at s, we can obtain a spanning tree by just disregarding the direction of the edges. As a result, if we have a procedure A that generates a random arborescence then, for a given spanning tree T, the probability that it will be generated is exactly $\sum_{s \in G} p_A(s)/|\mathcal{T}_s(G)| = \sum_{s \in G} p_A(s)/|\mathcal{T}(G)| = 1/|\mathcal{T}(G)|$. This means that if we interpret the arborescence returned by A as a spanning tree then we get in this way a random spanning tree.

7.1.2 An Outline of Our Approach

To describe our approach, let us consider a random walk X in G whose starting vertex is chosen according to the stationary distribution of G. If we just simulate X step-by-step until it covers the whole graph G, Theorem 7.1.1 asserts that from a transcript $X(\omega)$ of this simulation we can recover a random arborescence that is rooted at the starting vertex $s(\omega)$ of $X(\omega)$. However, while the whole transcript can have an expected length of $\Omega(mn)$, we utilize only a tiny fraction of entries—the $O(n)$ entries that allow us to recover arcs e_v for $v \in V \setminus \{s(\omega)\}$. It is thus natural to wonder whether the generation of the whole transcript is necessary. The random walk may spend long periods of time walking around regions of the graph that have already been covered, which seems quite wasteful. One may ask whether it is possible to identify and avoid such situations by somehow skipping such unnecessary steps. That is, one may ask whether there is a way of shortcutting the walk X such that the corresponding transcripts are much shorter, can be generated efficiently, and still retain the information that we need to recover the desired arborescence. We note that this intuition is quite reasonable for many of the standard examples of graphs that have large cover time, which consist of regions that are covered very quickly but in which the walk spends much of its time.

A tempting way to obtain such a shortcutting would be to try to just cut out from X all of its parts that correspond to visiting already explored parts of G. This shortcutting yields transcripts of length $O(n)$ and contains all of the information that we need. Unfortunately, it is not clear whether an efficient way of generating such transcripts exists—it is quite possible that the fastest way to find the next edge to an

[2]For brevity, we will hereafter omit the word "conditionally" when we refer to such an object. We stress that this is an object that we are introducing for technical reasons, and it should not be confused with the different problem of generating a uniformly random arborescence of a *directed* graph.

unvisited vertex traversed by the walk is to generate the whole trajectory of X inside the previously visited parts of G step-by-step.

The core of our approach is showing that there indeed exists an efficient way of shortcutting X. At a high level, the way we obtain it is as follows. We start by identifying a number of induced subgraphs $D_1, \ldots, D_k$ of G such that the cover time of each D_i is relatively small, and the set C of edges of G that are not inside any of the D_is constitutes a small fraction of the edges of G. Now, we shortcut the walk X in G by removing, for each i, the trajectories of X inside D_i that occur after X has already explored *the whole* D_i.[3] Such shortcut transcripts clearly retain all of the information that we need. Moreover, we show that the fact that D_is have small cover time and C has very small size imply that the expected length is also small. Finally, by exploiting the connection between random walks and electrical flows (namely, Fact 2.5.1) together with the linear system solver discussed in Section 2.7, we provide an efficient procedure to approximately generate such shortcut transcripts of X. All together, this yields the desired procedure for generating random arborescences.

7.1.3 Outline of This Chapter

The outline of this chapter is as follows. In section 7.2, we formally define the decompositions of G in which we are interested. We then relate the structure of the random walk X to these decompositions. Next, in Section 7.3, we show how these ideas can be used to develop an algorithm that generates (conditionally) random arborescence in expected time $\widetilde{O}(m^{3/2})$. Finally, in Section 7.4, we prove Theorem 7.1.2 by refining the previous algorithm to make it run in expected time $\widetilde{O}(m\sqrt{n})$.

7.2 The Structure of the Walk X

In this section, we shall formally define the decomposition of G that will be the basis of our algorithm and prove several important facts about the structure of the walk X with respect to this decomposition.

7.2.1 (ϕ, γ)-decompositions

While the algorithm sketched in the previous section could be made to use only edge cuts, our faster algorithm in Section 7.4 will require both vertex and edge cuts. To facilitate this, we shall use a decomposition that permits us to keep track of a both a set of edges C and a set of vertices S in the cut.

To this end, let $(D_1, \ldots, D_k, S, C)$ denote a partition of G such that $S \subseteq V(G)$, $\bigcup_i V(D_i) = V(G) \setminus S$, the D_i are disjoint induced subgraphs of G, and $C = E(G) \setminus \bigcup_i E(D_i)$ is the set of edges not entirely contained inside one of the D_i. For a given D_i, let $U(D_i)$ be the set of vertices of D_i incident to an edge from C, and let $C(D_i)$ be the subset of C incident to D_i.

[3] Later we modify this procedure slightly to get better running time for dense graphs.

Definition 7.2.1. *A partition* $(D_1, \ldots, D_k, S, C)$ *of* G *is a* (ϕ, γ)-decomposition *of* G *if:*

1. $|C| \leq \phi |E(G)|$,
2. *for each* i, *the diameter* $\gamma(D_i)$ *of* D_i *is less than or equal to* γ, *and*
3. *for each* i, $|C(D_i)| \leq |E(D_i)|$.

Note that this definition does not constrain the size of S explicitly, but $|S|$ is implicitly bounded by the fact that all edges incident to vertices of S are included in C.

Intuitively, the above decomposition corresponds to identifying in G induced subgraphs $D_1, \ldots, D_k$ that have diameter at most γ and contain all but a ϕ fraction of edges of G. We bound the diameters of the D_is and ensure (by the third condition) that they are not too "spread out" in G, since these properties will allow us to prove that X covers each of them relatively quickly. We will do this using the approach of Aleliunas *et al.* [5] that proved that the cover time of an unweighted graph G' with diameter $\gamma(G')$ is at most $O(|E(G')|\gamma(G'))$.

For now, let us assume that we have been given some fixed (ϕ, γ)-decomposition $(D_1, \ldots, D_k, S, C)$ of G (for some γ and ϕ that we will determine later). In Lemma 7.3.3, we will show that such decompositions with good parameters exist and can be constructed efficiently.

7.2.2 The Walk X

Let $X = (X_i)$ be a random walk in G that is started at a vertex chosen according to the stationary distribution of G, where X_i is the vertex visited in the i-th step. Let τ be the time corresponding to the first moment when our walk X has visited all of the vertices of G. Clearly, $E(\tau)$ is just the expected cover time of G, and by the fact that G has m edges and diameter at most n, the result of Aleliunas *et al.* [5] yields

Fact 7.2.2. $E[\tau] = O(mn)$.

Let Z be the random variable corresponding to the number of times that some edge from C was traversed by our random walk X, i.e., Z is the number of $i < \tau$ such that $X_i = v$, $X_{i+1} = v'$, and $(v, v') \in C$. Since, by Fact 7.2.2, the expected length of X is $O(mn)$, and we choose the starting vertex of X according to stationary distribution of G, the expected number of traversals of edges from C by X is just proportional to its size, so, on average, every $m/|C|$-th step of X corresponds to an edge from C. Therefore, the fact that in our decomposition $|C| \leq \phi |E(G)|$ implies

Fact 7.2.3. $E[Z] = O(\phi mn)$.

Let Z_i be the random variable corresponding to the number of times that some edge inside D_i is traversed by our walk. By definition, we have that $\tau = \sum_i Z_i + Z$. Now, for each D_i, let τ_i be the time corresponding to the first moment when we reach some vertex in $U(D_i)$ *after* all the vertices from D_i have been visited by our walk X.

Finally, let Z_i^* be the random variable corresponding to the number of times that some edge from $E(D_i)$ is traversed by our walk X until time τ_i occurs, i.e., until X explores the whole subgraph D_i. The following lemma holds:

Lemma 7.2.4. *For any i, $E[Z_i^*] = \widetilde{O}(|E(D_i)|\gamma(D_i))$.*

Before we proceed to the proof, it is worth noting that the above lemma does not directly follow from the result of Aleliunas *et al.*[5]. The reason is that [5] applies only to a natural random walk in D_i, and the walk induced by X in D_i is different. Fortunately, the fact that $|C(D_i)| \leq |E(D_i)|$ allows us to adjust the techniques of [5] to our situation and prove that a bound similar to the one of Aleliunas *et al.* still holds.

Proof. Let us fix $D = D_i$. For a vertex $v \in V(D)$, let $d_G(v)$ be the degree v in G, and let $d_D(v)$ be the degree of v in D. Clearly, $d_G(v) \geq d_D(v)$, and $d_C(v) \equiv d_G(v) - d_D(v)$ is the number of edges from C incident to v. For $u, v \in U(D)$, let $p_{u,v}^D$ be the probability that a random walk in G that starts at u will reach v through a path that does not pass through any edge inside D.

Consider a (weighted) graph D', which we obtain from D by adding, for each $u, v \in U(D)$, an edge (u, v) with weight $d_G(u) \cdot p_{u,v}^D$. (All edges from D have weight 1 in D'. Note that we do not exclude the case $u = v$, so we may add self-loops.) By the fact that the Markov chain corresponding to the random walk X in G is reversible, $d_G(u) \cdot p_{u,v}^D = d_G(v) \cdot p_{v,u}^D$, so our weights are consistent.

Now, the crucial thing to notice is that if we take our walk X and filter out of the vertices that are not from D, then the resulting "filtered" walk Y_D will just be a natural random walk in D' (with an appropriate choice of the starting vertex). In other words, the Markov chain corresponding to the random walk in D' is exactly the Markov chain induced on $V(D)$ by the Markov chain described by our walk X. As a result, since Z_i^* depends solely on the information that the induced random walk Y_D retains from X, it is sufficient to bound Z_i^* with respect to Y_D. However, it is easy to see that, in this case, $E[Z_i^*]$ can be upper-bounded by the expected time needed by a random walk in D', started at arbitrary vertex, to visit all of the vertices in D' and then reach some vertex in $U(D)$. We thus can conclude that $E[Z_i^*]$ is at most twice the cover time of D'. (More precisely, it is the cover time plus the maximum hitting time.)

Now, it is well-known (e.g., see [103]) that the cover time of any undirected graph G' is at most $2 \log |V(G')| H_{max}$, where $H_{max}(G')$ is the maximal hitting time in G'. Our aim is therefore to show that $H_{max}(D') = O(|E(D)|\gamma(D))$, where $\gamma(D)$ is the diameter of D. We will do this by applying the approach of Aleliunas *et al.*, who proved in [5] that for an *unweighted* graph G', $H_{max}(G') \leq |E(G')|\gamma(G')$.

To achieve this goal, let K be some number such that all weights in D' after multiplication by K become integral (which we can do since all of the weights in D' have to be rational). Let $K \cdot D'$ be the *unweighted* multigraph that we obtain from D' by multiplying all of its weights by K and then interpreting the new weights as multiplicities of edges. Note that the natural random walk in $K \cdot D'$ (treated as a sequence of visited vertices) is exactly the same as in D'.

Now, we prove that for two vertices v and w of D' such that an edge (v, w) exists in D, the expected time $H_{v,w}(K \cdot D) = H_{v,w}(D')$ until a random walk in $K \cdot D'$ that starts at v will reach w is at most $4|E(D)|$. To see this, note that the long-run frequency with which an copy of an edge is taken in a particular direction is $1/(2M)$, where M is total number of edges of $K \cdot D'$ (and we count each copy separately). Thus one of the K copies of edge (v, w) is taken in the direction from v to w every $K/(2M)$-th step on average. This in turn means that $H_{v,w}(D') \leq 2M/K$. Now, to bound M, we note first that $M \leq K(|E(D)| + \sum_{u,v \in U(D)} d_G(u) p^D_{u,v})$. Thus, since for a given $u \in U(D)$ the probability $\sum_v p^D_{u,v}$ of going outside D directly from u is equal to $d_C(u)/d_G(u)$, we obtain that $M \leq K(|E(D)| + \sum_u d_C(u)) \leq 2K|E(D)|$ by the property of a (ϕ, γ)-decomposition that requires that $|C(D)| \leq |E(D)|$. We can thus conclude that $H_{v,w}(D') \leq 2M/K \leq 4|E(D)|$, as desired. Having obtained this result, we can use a simple induction to show that $H_{v,w}(D') \leq 4|E(D)|\Delta(v, w)$, where $\Delta(v, w)$ is the distance between v and w in D. From this, we can conclude that $H_{max}(D') = 4|E(D)|\gamma(D)$ and $E[Z^*_i] \leq 16 \log n |E(D_i)|\gamma(D_i)$, as desired.

□

7.3 Our Algorithm

Let us now focus our attention on some particular D_i from our (ϕ, γ)-decomposition of G. The idea of our algorithm is based upon the following observation. Suppose we look at our random walk X just after time τ_i occurred. Note that this means that we already know for each $v \in V(D_i)$ which arc e_v we should add to the final arborescence. Therefore, from the point of view of building our arborescence, we gain no more information by knowing what trajectory X takes inside D_i after time τ_i. More precisely, if, at some step j, X enters D_i through some vertex $v \in V(D_i)$ and, after k steps, leaves through some edge $(u, u') \in C$, where $u \in V(D_i)$ and $u' \notin V(D_i)$, the actual trajectory $X_j, \ldots, X_{j+k}$ does not matter to us. The only point of simulating X inside D_i after time τ_i is to learn, upon entering D_i through v, through which edge $(u, u') \in C$ we should leave.

Let $P_v(e)$ be the probability of X leaving D_i through e after entering through vertex v. If we knew $P_v(e)$ for all $v \in V(D_i)$ and all $e \in C(D_i)$, then we could just, upon entering v, immediately choose the edge e through which we will exit according to distribution $P_v(e)$ without computing the explicit trajectory of X in D_i. That is, if we consider a shortcutting $\widetilde{X}$ of X that cuts out from X all trajectories inside D_i after it was explored, then $P_v(e)$ would be all that we need to simulate $\widetilde{X}$ in time proportional to the length of $\widetilde{X}$ (as opposed to the length of X).

Now, the point is that we can efficiently compute $P_v(e)$s to arbitrarily good precision, as we will show in Section 7.3.1, which in turn will enable us to simulate such a shortcut walk $\widetilde{X}$ sufficiently fast.

Furthermore, as we will see shortly, the analysis of structure of X that we performed in the previous section shows that the computation of these "unnecessary" trajectories constitutes the bulk of the work involved in a faithful simulation of X, and therefore $\widetilde{X}$ has much smaller length – and thus can be simulated much more

efficiently – while yielding the same distribution on random arborescences.

Therefore, in the light of the above discussion, all we really need to focus on now is showing that we can simulate a shortcut walk $\widetilde{X}$ as above within the desired time bounds (and that we can indeed find the underlying (ϕ, γ)-decomposition of G efficiently).

7.3.1 Simulation of the Shortcutting

We consider now the task of efficient simulation of the walk $\widetilde{X}$. Let us start by formalizing the description of the walk $\widetilde{X}$ that we provided above. To this end, let $\widetilde{X} = (\widetilde{X}_i)$ be a random walk obtained in the following way. Let $X(\omega) = X_0(\omega), \ldots X_{\tau(\omega)}(\omega)$ be some concrete trajectory of X, and let $X(\omega) = \overline{X}_1(\omega), \ldots, \overline{X}_{k(\omega)}(\omega)$ be decomposition of $X(\omega)$ into contiguous blocks $\overline{X}_j(\omega)$ that are contained inside D_{i_j} for some sequence $i_j \in \{0, \ldots, k\}$, where we adopt the convention that $D_0 = S$. We define $\widetilde{X}(\omega)$ as follows. We process $X(\omega)$ block-by-block and we copy a block $\overline{X}_j(\omega)$ to $\widetilde{X}(\omega)$ as long as τ_{i_j} has not occurred yet or $i_j = 0$, otherwise we copy to $\widetilde{X}(\omega)$ only the first and last entries of the block. (We shall refer to the latter event as a *shortcutting of the block*.)

Clearly, the only steps of the walk $\widetilde{X}$ that are non-trivial to simulate are the one corresponding to the shortcutting of the blocks. All the remaining steps are just the simple steps of a random walk in G. Recall that if we knew what are the probabilities $P_v(e)$ for each relevant vertex v and edge e, then these steps could be simulated efficiently too.

Unfortunately, it is not clear how to compute all of these probabilities within the desired time bounds. However, it turns out that we can use the connections between random walks, electrical flows, and fast Laplacian system solvers – as discussed in Chapter 2 – to compute a very good approximation of these probabilities fast. This is described in the following lemma.

Lemma 7.3.1. *We can compute additive δ-approximations of all of the probabilities $P_v(e)$ in time $\widetilde{O}(\phi m^2 \log 1/\delta)$.*

Proof. Let us fix some $D = D_i$ and an edge $e = (u, u') \in C(D)$ with $u \in U(D)$. Consider now a graph D' that we obtain from D as follows. First, we add vertex u', and some dummy vertex u^* to D, and then, for each $(w, w') \in C(D) \setminus \{e\}$ with $w \in U(D)$, we add an edge (w, u^*). (Note that w' can be equal to u'.) Finally, we add the edge $e = (u, u')$. The crucial thing to notice now is that for any given vertex $v \in D$, $P_v(e)$ is exactly the probability that a random walk in D' started at v will hit u' before it hits u^*. We can compute such probabilities quickly using electrical flows.

More precisely, if we treat D' as an electrical circuit and look at the vertex potentials $\boldsymbol{\phi}$ corresponding to the electrical u'-u^* flow of value 1 in D', then – by Fact 2.5.1 – the probability $P_v(e)$ will be equal to $\frac{\phi_v - \phi_{u^*}}{\phi_{u'} - \phi_{u^*}}$. Furthermore, by exploiting the fact that the task of finding such a vertex potentials $\boldsymbol{\phi}$ corresponds to solving a Laplacian system (cf. (2.10)), we can compute a δ-approximation of $P_v(e)$ for each $v \in V(D)$, in time $\widetilde{O}(|E(D')| \log 1/\delta)$ by applying the fast Laplacian system solver from Theorem

2.7.1. (Note that the error term $\varepsilon\|\boldsymbol{\phi}\|_{\boldsymbol{L}(D')} = \varepsilon\sqrt{\boldsymbol{\phi}^T \boldsymbol{L}(D')\boldsymbol{\phi}}$ in (2.16) is – by (2.12) – at most ε times the energy of the electrical u'-u^* flow of value 1 in D'. However, as all the resistances in D' are equal to 1, this energy can be only polynomial in n (in fact, at most n) and thus it suffices to take $\varepsilon = \delta/n^{O(1)}$ to get the desired accuracy.)

So, our algorithm performs the above computations for each edge e. (Note that we might have to two such computations per edge – one for each endpoint of e – if they each lie in different D_i.) From each computation, we store the probabilities $P_v(e)$ for all vertices v that we are interested in. The running time of this algorithm is bounded by $\widetilde{O}(|C|\sum_i |E(D_i)|\log 1/\delta) = \widetilde{O}(\phi m^2 \log 1/\delta)$, where we use the fact that for each D, $|E(D')| = |E(D)|+|C(D)| \leq 2|E(D)|$ by the definition of a (ϕ,γ)-decomposition. □

Since the algorithm from the above lemma gives us only δ-approximations to the values of the $P_v(e)$s, we need to show now that it is still sufficient for our purposes. Namely, in the following lemma we show how to utilize such an imperfect values of $P_v(e)$s to obtain a faithful simulation of the walk $\widetilde{X}$.

Lemma 7.3.2. *If one can obtain δ-approximate values of $P_v(e)$s in expected time $T(\phi,\gamma,m,n)\log 1/\delta$ then we can simulate the walk $\widetilde{X}$ (until it covers the whole graph) in expected time $\widetilde{O}(m(\gamma+\phi n)+T(\phi,\gamma,m,n))$.*

Note that if we plug for $T(\phi,\gamma,m,n)$ the bound offered by Lemma 7.3.1, the time of the simulation of the walk $\widetilde{X}$ would be $\widetilde{O}(m(\gamma+\phi n)+\phi m^2)$.

Proof. We start by bounding the expected length of the walk $\widetilde{X}$. By Fact 7.2.3 and Lemma 7.2.4, we have that this quantity is at most

$$\sum_i E[Z_i^*] + 3E[Z] = \widetilde{O}(m(\gamma+\phi n)), \tag{7.1}$$

where we amortized the number of shortcutting steps by tying them to the transitions over an edge in C that each of the shortcutting steps has to be followed or preceded by.

Now, we note that all we really need to do to simulate the walk $\widetilde{X}$ is to be able to obtain, for each $v \in \bigcup_i V(D_i)$, mn samples from the probability distribution corresponding to the probabilities $\{P_v(e)\}_e$. To see why this is the case, consider an algorithm that simulates up to mn first steps of the walk $\widetilde{X}$ using these samples and then simulates any further steps of $\widetilde{X}$ by just simulating the original walk X and shortcutting the blocks appropriately. Clearly, the expected cost of simulating the steps of $\widetilde{X}$ beyond the first mn steps, can be bounded by the expected length $E[\tau]$ of the walk X until it covers the whole graph, times the probability that the walk $\widetilde{X}$ will cover the whole graph in less than mn steps. By Fact 7.2.2, Markov's inequality and (7.1) we have that this cost is at most:

$$O(mn)\widetilde{O}(m(\gamma+\phi n))/mn = \widetilde{O}(m(\gamma+\phi n)),$$

in expectation and thus fits within our desired time bound.

In the light of the above, we just need to focus now on obtaining the above-mentioned samples efficiently enough.

To this end, let us first describe how these samples could be obtained if we somehow knew all the values $P_v(e)$. The way one could proceed then would be as follows. First, for each i and $v \in C(D_i)$, one would fix some ordering $e_1, \ldots, e_{|C(D_i)|}$ of the edges from $C(D_i)$. Next, for each $v \in U(D_i)$, one would create in $\widetilde{O}(|C(D_i)|)$ time an array A_v where $A_v(i) := \sum_{1 \leq j \leq i} P_v(e_j)$.

Clearly, the total time to compute the arrays A_v for all the relevant vs would be at most

$$\sum_i \widetilde{O}(|V(D_i)| \cdot |C(D_i)|) = \widetilde{O}(m|C|) = \widetilde{O}(\phi mn). \tag{7.2}$$

Now, to obtain the samples, one would choose a collection of random numbers r_v^l, for each $1 \leq l \leq mn$ and relevant v, where each r_v^l is chosen uniformly at random from the interval $[0, 1]$. Next, one would find for each v and l via binary search an entry j in $A_v(i)$ – for the corresponding i – such that $A_v(j) \geq r_v^l > A_v(j-1)$, and output $e_v^l := e_j$ as the sample corresponding to r_v^l. It is easy to verify that the obtained samples $\{e_v^l\}_{v,l}$ would have the desired distribution.

Obviously, the problem in implementing the above approach in our case is that Lemma 7.3.1 gives us access to only δ-approximate values of $P_v(e)$s. Therefore, if we repeated the above procedure with these values then the obtained entries $A_v(i)'$ in the arrays might differ by as much as $\delta i \leq \delta n$ from the true value of $A_v(i)$ and thus the obtained samples $\{e'^l_v\}_{v,l}$ might have different distribution than $\{e_v^l\}_{v,l}$.

The explain how we deal with this problem consider the situation when we have some $\{r_v^l\}_{v,l}$ chosen and are finding the corresponding edges $\{e'^l_v\}_{v,l}$ using the inaccurate values of $A_v(i)'$s instead of $A_v(i)$s. The key observation to make here is that as long as r_v^l chosen, for some v and l, differs by more than δn from all the entries in the array A'_v (we will call such a situation that the sample r_v^l is *clear*), then we know that the edge e'^l_v that we output is equal to the edge e_v^l that would be output if we used the true values of $A_v(i)$. In other words, whenever our sampling of r_v^l is clear, it does not matter that we are using only approximate values of $P_v(e)$s – the obtained sample is the same.

So, the way our sampling procedure will work is the following. Once the numbers $\{r_v^l\}_{v,l}$ are chosen, we find the smallest value k^* of k such that if we use the algorithm from Lemma 7.3.1 to compute the ε_k-approximate values of $P_v(e)$s with $\varepsilon_k = 1/2^{k+1}m^3n^2$, then all the r_v^ls are clear. We find such k^* by just starting with $k = 0$ and incrementing it each time there is some sample r_v^l that is still not clear. Once such k^* is found, it is easy to see that the samples $\{e_v^l\}_{v,l}$ produced from them using ε_{k^*}-approximate values of $P_v(e)$s have the desired distribution. So, the algorithm produces the right samples and indeed it provides the desired simulation of the walk $\widetilde{X}$.

Now, to bound its expected running time, we just need to bound the running time required to find k^* and compute the final samples $\{e'^l_v\}$ corresponding to it. To this end, note that the cost of computing the ε_k-approximate values of $P_v(e)$s and the

corresponding edge samples is, by Lemma 7.3.1 and (7.2), at most

$$\widetilde{O}(\phi mn) + T(\phi, \gamma, m, n) \log 1/\varepsilon_k = \widetilde{O}(T(\phi, \gamma, m, n)k + \phi mn).$$

On the other hand, the probability that for a given v and l the sample r_v^l differs by less than $\varepsilon_k n$ from a given $A_v(i)'$ is at most $4\varepsilon_k n$, as $A_v(i)'$ is within $\varepsilon_k n$ of the true value of $A_v(i)$ and thus r_v^l would need to fall within the interval $[A_v(j) - 2\varepsilon_k n, A_v(j) + 2\varepsilon_k n]$ for this to happen.

So, by union-bounding over all (at most m) values of i we get that the probability of r_v^l being not clear is at most $\varepsilon_k mn$. Finally, by union-bounding over all v and l, we get that a probability that there is at least one r_v^l that is not clear is at most

$$4\varepsilon_k m^3 n^2 \leq 2^{k-1}.$$

As a result, the total expected time of obtaining the needed edge samples is at most

$$\sum_{k=0}^{\infty} 2^{k-1} \widetilde{O}(T(\phi, \gamma, m, n)k + \phi mn) = \widetilde{O}(T(\phi, \gamma, m, n) + \phi mn),$$

as desired. The lemma follows. □

We now proceed to the final ingredient of our algorithm—finding good (ϕ, γ)-decompositions quickly.

7.3.2 Obtaining a Good (ϕ, γ)-decompositions Quickly

By using the ball-growing technique of Leighton and Rao [101] one can obtain the following result

Lemma 7.3.3. *For any G and any $\phi = o(1)$, there exists a $(\phi, \widetilde{O}(1/\phi))$-decomposition of G. Moreover, such a decomposition can be computed in $\widetilde{O}(m)$.*

As we will We omit the proof, as we shall prove a stronger statement in Lemma 7.4.2.

7.3.3 Generating a Random Arborescence in Time $\widetilde{O}(m^{3/2})$

We can now put these results together to generate a random arborescences in expected time $\widetilde{O}(m^{3/2})$. Note that this bound is worse than the one stated in Theorem 7.1.2—we shall improve it to obtain the better time bound in Section 7.4.

Let us set $\phi = 1/m^{1/2}$. By Lemma 7.3.3, we can get a $(1/m^{1/2}, \widetilde{O}(m^{1/2}))$-decomposition of G in $\widetilde{O}(m)$ time. As we explained already, all that we need to sample the desired arborescence is to be able to simulate the walk $\widetilde{X}$ until it covers the whole graph. By Lemma 7.3.1 and Lemma 7.3.2 this can be done in $\widetilde{O}(m(\gamma + \phi n) + \phi m^2) = \widetilde{O}(m^{3/2})$ time.

7.4 Obtaining an $\widetilde{O}(m\sqrt{n})$ Running Time

In this section, we show how to modify the $\widetilde{O}(m^{3/2}$ running time algorithm obtained above to make it run in $\widetilde{O}(m\sqrt{n})$ time and thus prove Theorem 7.1.2.

To understand what needs to modified to get this improvement, one should note that the main bottleneck in the algorithm presented above is the computation of the approximations to probabilities $P_v(e)$, i.e., Lemma 7.3.1 —everything else can be done in time $\widetilde{O}(m\sqrt{n})$ (if one would choose $\phi = 1/\sqrt{n}$). Unfortunately, it is not clear how we could improve the running time of these computations. To circumvent this problem, we will alter our algorithm to use slightly different probabilities and a slightly different random walk that will end up yielding a faster simulation time.

To introduce these probabilities, let us assume that there is no edges in G between different D_is (we will ensure that this is the case latter) and let $C_i(u)$ for a vertex $u \in S$ be the set of edges incident both to u and the component D_i. Now, for a given i, some $v \in V(D_i)$, and $u \in S$ with $|C_i(u)| > 0$, we define $Q_v(u)$ to be the probability that u is the first vertex not in $V(D_i)$ that is reached by a random walk that starts at v. We will use these probabilities to simulate a new random walk $\widehat{X}$ that we are about to define. For given trajectory $X(\omega)$ of walk X, $\widehat{X}(\omega)$ is equal to $\widetilde{X}(\omega)$ (as defined before) except that whenever $\widetilde{X}(\omega)$ shortcuts some block visited by X, $\widehat{X}(\omega)$ contains only the first vertex visited in this block (as opposed to both first and last vertices retained in $\widetilde{X}(\omega)$). Clearly, $\widehat{X}$ is a certain shortcutting of $\widetilde{X}$ and is thus a shortcutting of X as well.

It is not hard to see that by using of $Q_v(u)$ in a way completely analogous to the way we used $P_v(e)$ before, we can simulate the walk $\widehat{X}$ efficiently, and the expected length of this walk is bounded by the expected length of the walk $\widetilde{X}$. However, unlike $\widetilde{X}$, $\widehat{X}$ does not necessarily posses all of the information needed to reconstruct the final arborescence. This shortcoming manifests itself whenever some u is visited for the first time in $\widehat{X}$ directly after $\widehat{X}$ entered some D_i after τ_i has already occurred. In this case, we know that the corresponding trajectory of the walk X visited u for the first time through some edge whose other end was in D_i (and thus we should add it to our arborescence as e_u), but we don't know which one it was.

To deal with this problem, we will define a stronger decomposition of G whose properties will imply that the above failure to learn e_u will occur only for a small number of vertices u. Then, at the end of our simulation of the walk $\widehat{X}$, we will employ a procedure that will compute the missing arcs in a manner that will not distort the desired distribution over output arborescences.

We proceed now to formalizing the above outline.

7.4.1 Strong (ϕ, γ)-decompositions

The number of probabilities $Q_v(u)$ that we need to compute now depends on the number of vertices that are connected to the D_is through some edge from C, rather than just the number of edges of C. To control this, we introduce the following stronger graph decomposition:

Definition 7.4.1. *A partition $(D_1, \ldots, D_k, S, C)$ of G is a* strong (ϕ, γ)-decomposition *of G if:*

1. *$(D_1, \ldots, D_k, S, C)$ is a (ϕ,γ)-decomposition,*
2. *there is no edges between different D_is (i.e. S is a vertex multiway cut),*
3. *$|C(S)| \leq \phi|V(G)|$, where $C(S)$ is the set of vertices from S that are connected by an edge to some D_i.*

We prove the following

Lemma 7.4.2. *For any G, and any $\phi = o(1)$ there exists a strong $(\phi, \widetilde{O}(1/\phi))$-decomposition of G. Moreover, such a decomposition can be computed in $\widetilde{O}(m)$.*

Proof. We will use a slightly modified version of the ball-growing technique of Leighton and Rao [101] as presented by Trevisan [134]. For a graph H, let $B_H(v, j)$ be the ball of radius j around the vertex v in H, i.e., let $B_H(v, j)$ consist of the subgraph of H induced by all vertices in H that are reachable by a path of length at most j from v. Furthermore, let $R_H(v, j)$ be the set of vertices that are at distance exactly i from v in H. Finally, let $R^+_H(v, j)$ ($R^-_H(v, j)$ respectively) be the set $E(B_H(v, j+1)) \setminus E(B_H(v, j))$ (the set $E(B_H(v, j)) \setminus E(B_H(v, j-1))$ respectively).

Consider now the following procedure:

- Set $H = G$, $S = \emptyset$, $C = \emptyset$, $D = \{\}$, $i = 1$ and $t = \phi/(1-\phi)$
- While $H \neq \emptyset$
- (* Ball-growing *)
  - Choose an arbitrary $v \in H$, set $j = 0$
  - (*) As long as $|R_H(v, j+1)| > t|V(B_H(v, j))|$, $|R^+_H(v, j+1)| > t|E(B_H(v, j))|$ or $|R^-_H(v, j+1)| > t|E(B_H(v, j))|$:
  - $j = j + 1$
  - Let j_i be the j at which the above loop stops. Add $R_H(v, j_i + 1)$ to S, add all the edges incident to $R_H(v, j_i + 1)$ (i.e. $R^+_H(v, j_i + 1) \cup R^-_H(v, j_i + 1)$) to C, and add $B_H(v, j_i)$ as component D_i to D.
- output the resulting partition $(D_1, \ldots, D_k, S, C)$ of G

First, we note that the above procedure can be implemented in nearly-linear time, since each edge is examined at most twice before it is removed from H. Moreover, an elementary charging argument shows that in the resulting partition $|C| \leq (1/(1 + 1/t))|E(G)| = \phi|E(G)|$, and similarly $|C(S)| = |S| \leq \phi|V(G)|$. By construction, there is no edges between distinct D_is. We want to argue now that for all i, $j_i \leq 3(1 + \log|E(G)|/\log(1 + t))$, which in turn would imply that all the D_is have diameter at most $3(1 + \log|E(G)|)/\log(1 + t) = 3(1 + \log|E(G)|)/\log(1/(1 - \phi))) =$

$O(\log m/(-\log(1-\phi))) = O(\log m/\phi)$, where we used Taylor expansion of $\log(1-x)$ around $x = 0$ to get this estimate. To see why the above bound on j_i holds, assume that it was not the case for some i and v. Then, during the corresponding ball-growing procedure, a particular one of the three conditions from $(*)$ must have been triggered more than $j_i/3 = 1 + \log|E(G)|/\log(1+t)$ times. If this condition was $|R_H(v, j+1)| > t|V(B_H(v, j))|$, then, since we never remove vertices from our ball that is being grown and $B_H(v, 0)$ has one vertex, the final ball $B_H(v, j_i)$ has to have at least $(1+t)^{j_i/3} > |E(G)| \geq |V(G)|$ vertices, which is a contradiction. Similarly, if $|R^+_H(v, j+1)| > t|E(B_H(v, j))|$ ($|R^-_H(v, j+1)| > t|E(B_H(v, j))|$ respectively) was the condition in question, then $|E(B_H(j_i, v))| > (1+t)^{j_i/3} \geq |E(G)|$, which is a contradiction as well. Thus we may conclude that the above bound on j_i holds, and thus all D_is have diameter $\widetilde{O}(1/\phi)$, as desired.

At this point, we know that the partition of G that we obtained satisfies all of the properties of a strong $(\phi, \widetilde{O}(1/\phi))$-decomposition except possibly the one that asserts that there is no D_i such that $|E(D_i)|$ is smaller than the number $|C(D_i)|$ of edges from C incident to D_i. However, if such D_i exist then we can just add them to our cut, i.e., we add $V(D_i)$ to S, and $E(D_i)$ to C. Note that the size of C can at most triple as a result of this operation, since an edge that is initially in C can be incident to at most two of the D_i, and edges $E(D_i)$ are by definition not incident to any other D_j. Similarly, $C(S)$ does not increase as a result of adding $V(D_i)$ to S. Thus, we may conclude that the decomposition returned by this algorithm is indeed a strong $(\phi, \widetilde{O}(1/\phi))$-decomposition of G.

□

From now on we fix some *strong* (ϕ, γ)-decomposition of G.

7.4.2 Computating of $Q_v(u)$

By using the similar approach to the one that we used when computing the $P_v(e)$ values, we get the following lemma.

Lemma 7.4.3. *Given a strong (ϕ, γ)-decomposition of G, we can compute an additive δ-approximations of all $Q_v(u)$ in time $\widetilde{O}(\phi mn \log 1/\delta)$.*

Proof. Let us fix some $D = D_i$, let S' be the set of vertices in $w \in S$ with $|C_i(w)| > 0$, and let us fix some $u \in S'$. Consider now a graph D'_u that we obtain from $D \cup S'$ by merging all vertices in $S' \setminus \{u\}$ into one vertex called u^*. For any given vertex $v \in D$, $Q_v(u)$ is exactly the probability that a random walk in D'_u started at v will hit u before it hits u^*. Once again, as in Lemma 7.3.1, we can quickly compute such probabilities using the connection of random walks to electrical flows (cf. Fact 2.5.1) and fast Laplacian system solver as discussed in Section 2.7 – see the proof of Lemma 7.3.1 for details.

Note that we will need to run Laplacian system solver per each vertex u in $C(S)$ and a component D_i such that $|C_i(u)| > 0$. Thus the total running time of the resulting algorithm is bounded by $\widetilde{O}(|C(S)| \sum_i |E(D_i)| \log 1/\delta) = \widetilde{O}(\phi mn \log 1/\delta)$ as desired. □

7.4.3 Coping with Shortcomings of $\widehat{X}$

As mentioned above, the walk $\widehat{X}$ can be simulated more efficiently than the walk $\widetilde{X}$, but it does not have all the information needed to construct the arborescence that would be generated by the walk X that $\widehat{X}$ is meant to shortcut. This lack of information occurs only when some vertex u is visited for the first time by $\widehat{X}$ immediately after $\widehat{X}$ visited a component D_i after time τ_i has already occurred. Note that, by the properties of the strong (ϕ, γ)-decomposition that we are using, it must be the case that $u \in C(S)$ and $|C(S)| \leq \phi n$. This shows that $\widehat{X}$ fails to estimate the arcs e_u for a small fraction of vertices of G. We prove now that, in this case, these missing arcs can be reconstructed efficiently and in a way that preserves the desired distribution over arborescences.

Lemma 7.4.4. *For a trajectory $\widehat{X}(\omega)$ that starts at vertex s, let $F(\widehat{X}(\omega))$ be the set of arcs e_v for $v \notin C(S) \cup \{s\}$ corresponding to this walk. Let F^* be the set of all arborescences H rooted at s such that $e_H(v) = e_v$ for $v \notin C(S) \cup \{s\}$. Then, given F, we can generate a random arborescence from F^* in time $O(m + (\phi n)^{2.376})$.*

Proof. Let $H_1, \ldots, H_r$, $s \in H_1$ be the decomposition of F into weakly connected components. We construct a *directed* graph $G(F, s)$ as follows. $G(F, s)$ has a vertex h_j for each H_j. $E(G(F, s))$ is the set of all arcs (h_j, h_l) such that there exists $v, u \in G$ with $v \in H_j$, $u \in H_l$ and $u \in C(S) \setminus \{s\}$. By definition, if H' is an arborescence in $G(F, s)$ rooted at h_1, then $H' \cup F$ is an arborescence in G rooted at s and $H' \cup F \in F^*$. Moreover, for any arborescence $H \in F^*$, $H' = \{e_H(v) \mid v \in C(S) \setminus \{s\}\}$ is an arborescence in $G(F, s)$ rooted at h_1. So, if we use the algorithm of Colbourn *et al.* [45] to generate random arborescence H' in $G(F, s)$ rooted at h_1, then $H' \cup F$ is a random arborescence from F^*. Since $|V(G(F, s))| = |C(S)| \leq \phi n$ and the algorithm from [45] works in time $O(|V(G(F, s))|^{2.376})$, the lemma follows.

□

7.4.4 Proof of Theorem 7.1.2

By the connection explained in the introduction (section 7.1.1), it is sufficient to devise a procedure that generates a random arborescences. We do this as follows. We fix $\phi = 1/\sqrt{n}$, and using Lemma 7.4.2 we get a strong $(1/\sqrt{n}, \widetilde{O}(\sqrt{n}))$-decomposition of G in $\widetilde{O}(m)$ time. Now, we use completely analogous reasoning to the one used in Lemma 7.3.2 – together with the algorithm offered by Lemma 7.4.3 – to prove that we can simulate (δ-approximately) the walk $\widehat{X}$ in expected time $\widetilde{O}(m\sqrt{n})$ (we use in particular the fact that $\widehat{X}$ is a shortcutting of $\widetilde{X}$ and thus of X). Having done this, we look at the directed forest F consisting of arcs e_v for $v \notin C(S) \cup \{s\}$ as defined by our simulation of $\widehat{X}$. We use the procedure from Lemma 7.4.4 to get an arborescence T of G in time $O(m + n^{2.376/2}) = O(m\sqrt{n})$. The whole algorithm therefore runs in expected time $\widetilde{O}(m\sqrt{n})$.

To see that the above algorithm generates a (conditionally) random arborescence, let us consider some arborescence T of G rooted at some vertex s and condition what will follow on event that the algorithm outputs an arborescence rooted at s.

Let $F(T) = \{e_T(v) \mid v \notin C(S) \cup \{s\}\}$. By the fact that $\widehat{X}$ is a a shortcutting of the walk X and by Theorem 7.1.1, we know that $F(\widehat{X}(\omega)) = F(T)$ for exactly $|F(T)^*|/|\mathcal{T}_s(G)|$ fraction of trajectories. But since the procedure from Lemma 7.4.4 generates a random arborescence from $F(T)^*$, T is generated with probability exactly $1/|\mathcal{T}_s(G)|$. This concludes the proof of Theorem 7.1.2.

Chapter 8

Conclusions

In this thesis, we developed new approaches to obtaining fast approximation algorithms for a variety of fundamental graph problems. These approaches draw upon a diverse set of modern algorithmic techniques such as sparsification, low-stretch spanning trees, the multiplicative-weights-update method, and dynamic graph algorithms; as well as, fast Laplacian system solvers and various tools of spectral graph theory. The paradigm that played an especially key role in establishing our results was connecting combinatorial properties of graphs to linear-algebraic properties of associated Laplacians. In particular, we were using electrical flows and the related objects to probe the cut/flow structure of the underlying graph.

These ideas led not only to obtaining faster algorithms for cut- and flow-based graph problems. They also allowed us to understand better and make progress on a basic optimization task: approximation of the asymmetric traveling salesman problem; as well as, provided us with a faster algorithm for a basic algorithmic problem of spectral graph theory: the generation of an uniformly random spanning tree.

We believe that the research presented in this thesis opens up a lot of interesting directions to pursue. We survey some of them below.

Maximum s-t Flow Problem: Given the fundamental nature of the maximum s-t flow problem, it is natural to ask whether one can refine further the approach we presented in Chapter 3, to make it give us an algorithm for the undirected maximum s-t flow problem that runs in nearly-linear time and computes an exact answer. Note that by the connection we described in Section 3.6, having such an algorithm for undirected graphs would also yield a nearly-linear time algorithm for the directed setting.

One observation to keep in mind when investigating the above question is that it is known [14, 95] that the multiplicative-weight-update routine we described in Section 2.8 and employed in Chapter 3, is fundamentally unable to guarantee better than polynomial in $1/\varepsilon$ dependence of the running time on the error parameter ε. Therefore, to make our electrical-flow-based approach lead to exact answers, one would need to change the way the electrical flow computations are combined to form the final answer. One promising approach to doing that is to use the ideas underlying the interior point methods. It is known that the running time of the generic interior

point method algorithms can be sped up significantly in the context of flow problems [51]. So, one could wonder if it is possible to obtain further improvement here by utilizing the insight gained from understanding the interplay between electrical flows and the combinatorial structure of graphs.

Unifying the Landscape of Algorithmic Graph Theory: As our ultimate goal is to develop fast and well-performing algorithms for all the fundamental graph problems, one could consider taking a more principled approach towards this goal than just designing algorithms for each particular problem separately.

One way of pursuing this direction might be by trying to develop a general method of obtaining efficient algorithms for some broad family of graph problems that is similar in spirit to our framework from Chapter 5. Ideally, such a method would allow us to reduce the task of designing good algorithms for general instances of the problems in this family, to a task of designing good algorithms for instances of these problems that correspond to some structurally simpler class of graphs (e.g., trees). One concrete example where obtaining such a result might be possible, would be extending the framework from Chapter 5 to flow-based problems.

Another approach that might lead to a more unified picture of the structure of close-to-linear-time graph algorithms is based on reducibility. More precisely, in the spirit of the theory of NP-completeness, one could try to relate various algorithmic tasks to each other by establishing formal reductions between them. These reductions would aim at examining how fast a given task can be performed if one has an efficient algorithm that solves some other task.

A particular direction in this vein that one could pursue is tying the complexity of various fundamental graph problems to the complexity of the maximum s-t flow problem. It is already known that a variety of optimization tasks (e.g., bipartite matching and graph partitioning [123]) can be reduced to maximum s-t flow computations. Therefore, the hope would be that by following this direction more systematically, we would establish the maximum s-t flow problem as the central problem of the algorithmic graph theory. In this case, we could focus our attention on understanding this particular problem, as developing a fast algorithm for it would immediately imply a fast algorithms for all the tasks that were reduced to it.

Further Study of the Connections Between the Combinatorial Algorithms and Linear Algebra: Given the presented in this thesis benefits of integration of linear-algebraic and combinatorial views on graphs, a natural direction to pursue is understand further the extent to which this paradigm can be applied. In particular, one might try to examine the known applications of the traditional spectral methods and investigate whether the additional insight granted by the usage of electrical flows leads to some improvements there.

For example, one of the classical applications of spectral graph theory is using Cheeger's Inequality (cf. Section 2.5.1) to obtain efficient graph partitioning primitives. The performance of these primitives is very good on graphs that have very good connectivity properties, i.e., on graphs having fairly large conductance, but de-

grades quickly as this connectivity worsens. It would be interesting to understand if basing the partitioning procedure on electrical flows – instead of basing it just on the eigenvector corresponding to the second-smallest eigenvalue of the Laplacian – could mitigate this problem.

Yet another possible application of electrical flows might be making further progress on understanding thin trees. As explained in Chapter 6, obtaining $(\alpha, O(1))$-thin trees, for $\alpha = o(\log n / \log \log n)$, would allow us to obtain an improved, $O(\alpha)$ factor, approximation to the asymmetric traveling salesman problem. As electrical-flow-based techniques were already useful in establishing our $(O(\log n / \log \log n), O(1))$-thinness bound, it is possible that a deeper understanding of the interplay of electrical flows and the cut structure of graphs will lead to further improvement.

Bibliography

[1] I. Abraham, Y. Bartal, and O. Neiman. Nearly tight low stretch spanning trees. In *FOCS'08: Proceedings of the 49th Annual IEEE Symposium on Foundations of Computer Science*, pages 781–790, Washington, DC, USA, 2008. IEEE Computer Society. Full version available at arXiv:0808.2017.

[2] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network flows: theory, algorithms, and applications.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[3] R. K. Ahuja, T. L. Magnanti, J. B. Orlin, and M. R. Reddy. *Applications of network optimization.* North-Holland, 1995.

[4] D. J. Aldous. A random walk construction of uniform spanning trees and uniform labelled trees. *SIAM Journal on Discrete Mathematics*, 3(4):450–465, 1990.

[5] R. Aleliunas, R. M. Karp, R. J. Lipton, L. Lovász, and C. Rackoff. Random walks, universal traversal sequences, and the complexity of maze problems. In *FOCS'79: Proceedings of the 20th Annual IEEE Symposium on Foundations of Computer Science*, pages 218–223, 1979.

[6] N. Alon. Eigenvalues and expanders. *Combinatorica*, 38(1):83–96, 1986.

[7] N. Alon, Z. Galil, O. Margalit, and M. Naor. Witnesses for boolean matrix multiplication and for shortest paths. In *FOCS '92: Proceedings of the 33rd Annual IEEE Symposium on Foundations of Computer Science*, pages 417–426, Washington, DC, USA, 1992. IEEE Computer Society.

[8] N. Alon, R. M. Karp, D. Peleg, and D. West. A graph-theoretic game and its application to the k-server problem. *SIAM Journal on Computing*, 24(1):78–100, 1995.

[9] N. Alon and V. Milman. λ_1, isoperimetric inequalities for graphs and superconcentrators. *Journal of Combinatorial Theory, Series B*, 38:73–88, 1985.

[10] I. Althöfer, G. Das, D. Dobkin, D. Joseph, and J. Soares. On sparse spanners of weighted graphs. *Discrete and Computational Geometry*, 9:81–100, January 1993.

[11] R. Andersen, F. Chung, and K. Lang. Local graph partitioning using pagerank vectors. In *FOCS'06: Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*, pages 475–486, Washington, DC, USA, 2006. IEEE Computer Society.

[12] R. Andersen and Y. Peres. Finding sparse cuts locally using evolving sets. In *STOC'09: Proceedings of the 41st Annual ACM symposium on Theory of Computing*, pages 235–244, New York, NY, USA, 2009. ACM.

[13] S. Arora, E. Hazan, and S. Kale. $O(\sqrt{\log n})$ approximation to sparsest cut in $\widetilde{O}(n^2)$ time. In *FOCS'04: Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science*, pages 238–247, Washington, DC, USA, 2004. IEEE Computer Society.

[14] S. Arora, E. Hazan, and S. Kale. The multiplicative weights update method: A meta-algorithm and applications. Available at http://www.cs.princeton.edu/~arora/pubs/MWsurvey.pdf, 2005.

[15] S. Arora and S. Kale. A combinatorial, primal-dual approach to semidefinite programs. In *STOC'07: Proceedings of the 39th Annual ACM Symposium on Theory of Computing*, pages 227–236, New York, NY, USA, 2007. ACM.

[16] S. Arora, J. R. Lee, and A. Naor. Euclidean distortion and the sparsest cut. In *STOC'05: Proceedings of the 37th Annual ACM Symposium on Theory of Computing*, pages 553–562, New York, NY, USA, 2005. ACM.

[17] S. Arora, S. Rao, and U. Vazirani. Expander flows, geometric embeddings and graph partitioning. *Journal of the ACM*, 56(2):1–37, 2009.

[18] A. Asadpour, M. Goemans, A. Mądry, S. Oveis Gharan, and A. Saberi. An $O(\log n)$-approximation algorithm for the asymmetric traveling salesman problem. In *SODA'10: Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 379–389, 2010.

[19] Y. Aumann and Y. Rabani. An O(log k) approximate min-cut max-flow theorem and approximation algorithm. *SIAM Journal on Computing*, 27(1):291–301, 1998.

[20] G. Ausiello, G. F. Italiano, A. M. Spaccamela, and U. Nanni. On-line computation of minimal and maximal length paths. *Theoretical Computer Science*, 95(2):245–261, 1992.

[21] Y. Azar, E. Cohen, A. Fiat, H. Kaplan, and H. Räcke. Optimal oblivious routing in polynomial time. In *STOC'03: Proceedings of the 35th Annual ACM Symposium on Theory of Computing*, pages 383–388, New York, NY, USA, 2003. ACM.

[22] S. Baswana, R. Hariharan, and S. Sen. Maintaining all-pairs approximate shortest paths under deletion of edges. In *SODA'03: Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 394–403, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[23] S. Baswana, R. Hariharan, and S. Sen. Improved decremental algorithms for maintaining transitive closure and all-pairs shortest paths. *Journal of Algorithms*, 62(2):74–92, 2007.

[24] S. Baswana, T. Kavitha, K. Mehlhorn, and S. Pettie. New constructions of (α, β)-spanners and purely additive spanners. In *SODA '05: Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 672–681, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.

[25] J. D. Batson, D. A. Spielman, and N. Srivastava. Twice-Ramanujan sparsifiers. In *STOC'09: Proceedings of the 41st Annual ACM symposium on Theory of Computing*, pages 255–262, New York, NY, USA, 2009. ACM.

[26] A. A. Benczúr and D. R. Karger. Approximating s-t minimum cuts in $\widetilde{\mathrm{O}}(n^2)$ time. In *STOC'96: Proceedings of the 28th Annual ACM Symposium on Theory of Computing*, pages 47–55, New York, NY, USA, 1996. ACM.

[27] A. A. Benczúr and D. R. Karger. Randomized approximation schemes for cuts and flows in capacitated graphs. *CoRR*, cs.DS/0207078, 2002.

[28] M. Bienkowski, M. Korzeniowski, and H. Räcke. A practical algorithm for constructing oblivious routing schemes. In *SPAA'03: Proceedings of the 15th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 24–33, New York, NY, USA, 2003. ACM.

[29] D. Bienstock and G. Iyengar. Solving fractional packing problems in $\widetilde{O}(1/\varepsilon)$ iterations. In *STOC'04: Proceedings of the 36th Annual ACM Symposium on Theory of Computing*, pages 146–155, New York, NY, USA, 2004. ACM.

[30] M. Bläser. A new approximation algorithm for the asymmetric TSP with traingle inequality. In *SODA*, pages 638–645, 2002.

[31] B. Bollobas. *Graph Theory: An Introductory Course*. Springer-Verlag, 1979.

[32] B. Bollobas. *Modern Graph Theory*. Springer, 1998.

[33] A. Broder. Generating random spanning trees. In *FOCS'89: Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science*, pages 442–447, 1989.

[34] P. Buergisser, M. Clausen, and A. Shokrollahi. *Algebraic Complexity Theory*. Grundlehren der mathematischen Wissenschaften. Springer Verlag, 1996.

[35] J. R. Bunch and J. E. Hopcroft. Triangular factorization and inversion by fast matrix multiplication. *Mathematics of Computation*, 28(125):231–236, 1974.

[36] M. Charikar, M. Goemans, and H. Karloff. On the integrality ratio for the asymmetric traveling salesman problem. *Mathematics of Operations Research*, 31:245–252, 2006.

[37] S. Chawla, A. Gupta, and H. Räcke. Embeddings of negative-type metrics and an improved approximation to generalized sparsest cut. In *SODA'05: Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 102–111, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.

[38] J. Cheeger. A lower bound for the smallest eigenvalue of the laplacian. In *Problems in Analysis, eds Gunning RC*, 1970.

[39] C. Chekuri, J. Vondrak, and R. Zenklusen. Dependent randomized rounding via exchange properties of combinatorial structures. In *FOCS'10: Proceedings of the 51st Annual IEEE Symposium on Foundations of Computer Science*, pages 575 –584, oct. 2010.

[40] P. Christiano, J. Kelner, A. Mądry, D. Spielman, and S.-H. Teng. Electrical flows, Laplacian systems, and faster approximation of maximum flow in undirected graphs. In *STOC'11: Proceedings of the 43rd Annual ACM Symposium on Theory of Computing*, pages 273–281, 2011.

[41] N. Christofides. Worst case analysis of a new heuristic for the traveling salesman problem. Report 388, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, PA, 1976.

[42] F. R. K. Chung. *Spectral Graph Theory*. American Mathematical Society, 1997.

[43] C. J. Colbourn, R. P. J. Day, and L. D. Nel. Unranking and ranking spanning trees of a graph. *Journal of Algorithms*, 10(2):271–286, 1989.

[44] C. J. Colbourn, B. M. Debroni, and W. J. Myrvold. Estimating the coefficients of the reliability polynomial. *Congressus Numerantium*, 62:217–223, 1988.

[45] C. J. Colbourn, W. J. Myrvold, and E. Neufeld. Two algorithms for unranking arborescences. *Journal of Algorithms*, 20(2):268–281, 1996.

[46] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9:251–280, 1990.

[47] G. Călinescu, H. Karloff, and Y. Rabani. An improved approximation algorithm for multiway cut. In *STOC '98: Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 48–52, New York, NY, USA, 1998. ACM.

[48] W. H. Cunningham and L. Tang. Optimal 3-terminal cuts and linear programming. In *IPCO '99: Proceedings of the 7th International IPCO Conference on Integer Programming and Combinatorial Optimization*, pages 114–125, London, UK, 1999. Springer-Verlag.

[49] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. The complexity of multiway cuts (extended abstract). In *STOC '92: Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 241–251, New York, NY, USA, 1992. ACM.

[50] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. The complexity of multiterminal cuts. *SIAM Journal on Computing*, 23(4):864–894, 1994.

[51] S. I. Daitch and D. A. Spielman. Faster approximate lossy generalized flow via interior point algorithms. In *STOC'08: Proceedings of the 40th Annual ACM Symposium on Theory of Computing*, pages 451–460, 2008.

[52] C. Demetrescu and G. F. Italiano. Fully dynamic transitive closure: breaking through the $o(n^2)$ barrier. In *FOCS'00: Proceedings of the 41st Annual IEEE Symposium on Foundations of Computer Science*, page 381, Washington, DC, USA, 2000. IEEE Computer Society.

[53] C. Demetrescu and G. F. Italiano. A new approach to dynamic all pairs shortest paths. In *STOC'03: Proceedings of the 35th Annual ACM Symposium on Theory of Computing*, pages 159–166, New York, NY, USA, 2003. ACM.

[54] C. Demetrescu and G. F. Italiano. Fully dynamic all pairs shortest paths with real edge weights. *Journal of Computer and System Sciences*, 72(5):813–837, 2006.

[55] J. Dodziuk. Difference equations, isoperimetric inequality and transience of certain random walks. *Transactions of the American Mathematical Society*, 284(2):787–794, 1984.

[56] J. Edmonds. Matroids and the greedy algorithm. *Mathematical Programming*, pages 127–136, 1971.

[57] P. Elias, A. Feinstein, and C. E. Shannon. A note on the maximum flow through a network. *IRE Transactions on Information Theory*, 2, 1956.

[58] M. Elkin, Y. Emek, D. A. Spielman, and S.-H. Teng. Lower-stretch spanning trees. In *STOC'05: Proceedings of the 37th Annual ACM Symposium on Theory of Computing*, pages 494–503, New York, NY, USA, 2005. ACM.

[59] M. Elkin and D. Peleg. $(1+\epsilon, \beta)$-spanner constructions for general graphs. *SIAM Journal on Computing*, 33(3):608–631, 2004.

[60] S. Even and Y. Shiloach. An on-line edge-deletion problem. *Journal of the ACM*, 28(1):1–4, 1981.

[61] S. Even and R. E. Tarjan. Network flow and testing graph connectivity. *SIAM Journal on Computing*, 4(4):507–518, Dec. 1975.

[62] J. Fakcharoenphol, S. Rao, and K. Talwar. A tight bound on approximating arbitrary metrics by tree metrics. In *STOC'03: Proceedings of the 35th Annual ACM Symposium on Theory of Computing*, pages 448–455, New York, NY, USA, 2003. ACM.

[63] U. Feige and M. Singh. Improved approximation ratios for traveling salesman tours and paths in directed graphs. In *APPROX*, pages 104–118, 2007.

[64] M. Fiedler. Algebraic connectivity of graphs. *Czechoslovak Mathematical Journal*, 23(98):298–305, 1973.

[65] L. K. Fleischer. Approximating fractional multicommodity flow independent of the number of commodities. *SIAM Journal of Discrete Mathematics*, 13(4):505–520, 2000.

[66] R. W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962.

[67] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[68] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34:596–615, July 1987.

[69] A. M. Frieze, G. Galbiati, and F. Maffioli. On the worst-case performance of some algorithms for the asymmetric traveling salesman problem. *Networks*, 12:23–39, 1982.

[70] N. Garg and J. Könemann. Faster and simpler algorithms for multicommodity flow and other fractional packing problems. In *FOCS'98: Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science*, pages 300–309, 1998.

[71] M. X. Goemans. Minimum bounded degree spanning trees. In *FOCS'06: Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*, pages 273–282, 2006.

[72] A. V. Goldberg. A natural randomization strategy for multicommodity flow and related algorithms. *Information Processing Letters*, 42(5):249–256, 1992.

[73] A. V. Goldberg and S. Rao. Beyond the flow decomposition barrier. *Journal of the ACM*, 45(5):783–797, 1998.

[74] A. V. Goldberg and R. E. Tarjan. Finding minimum-cost circulations by canceling negative cycles. *Journal of the ACM*, 36:873–886, October 1989.

[75] N. Goyal, L. Rademacher, and S. Vempala. Expanders via random spanning trees. In *SODA'09: Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 576–585, 2009.

[76] M. D. Grigoriadis and L. G. Khachiyan. Fast approximation schemes for convex programs with many blocks and coupling constraints. *SIAM Journal on Optimization*, 4(1):86–107, 1994.

[77] M. D. Grigoriadis and L. G. Khachiyan. Approximate minimum-cost multicommodity flows in $\widetilde{O}(\varepsilon^{-2}knm)$ time. *Mathematical Programming*, 75(3):477–482, 1996.

[78] A. Guénoche. Random spanning tree. *Journal of Algorithms*, 4(3):214–220, 1983.

[79] N. S. H. Kaplan, M. Lewenstein and M. Sviridenko. Approximation algorithms for asymmetric TSP by decomposing directed regular multigraphs. *Journal of the ACM*, pages 602–626, 2005.

[80] J. Hao and J. B. Orlin. A faster algorithm for finding the minimum cut in a graph. In *SODA'92: Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '92, pages 165–174, Philadelphia, PA, USA, 1992. Society for Industrial and Applied Mathematics.

[81] C. Harrelson, K. Hildrum, and S. Rao. A polynomial-time tree decomposition to minimize congestion. In *SPAA'03: Proceedings of the 15th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 34–43, New York, NY, USA, 2003. ACM.

[82] M. Held and R. Karp. The traveling salesman problem and minimum spanning trees. *Operations Research*, 18:1138–1162, 1970.

[83] D. Kandel, Y. Matias, R. Unger, and P. Winkler. Shuffling biological sequences. *Discrete Applied Mathematics*, 71(1-3):171–185, 1996.

[84] G. Karakostas. Faster approximation schemes for fractional multicommodity flow problems. In *SODA'02: Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 166–173, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[85] D. Karger and S. Plotkin. Adding multiple cost constraints to combinatorial optimization problems, with applications to multicommodity flows. In *STOC'95: Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, pages 18–25, New York, NY, USA, 1995. ACM.

[86] D. R. Karger. Global min-cuts in rnc, and other ramifications of a simple min-out algorithm. In *SODA '93: Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 21–30, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics.

[87] D. R. Karger. Better random sampling algorithms for flows in undirected graphs. In *SODA '98: Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 490–499, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics.

[88] D. R. Karger. Minimum cuts in near-linear time. *Journal of the ACM*, 47:46–76, January 2000.

[89] D. R. Karger, P. Klein, C. Stein, M. Thorup, and N. E. Young. Rounding algorithms for a geometric embedding of minimum multiway cut. In *STOC '99: Proceedings of the 31st Annual ACM Symposium on Theory of Computing*, pages 668–678, New York, NY, USA, 1999. ACM.

[90] D. R. Karger and C. Stein. An $\tilde{O}(n^2)$ algorithm for minimum cuts. In *STOC'93: Proceedings of the 25th Annual ACM Symposium on the Theory of Computing*, pages 757–765, New York, NY, USA, 1993. ACM.

[91] J. Kelner and A. Mądry. Faster generation of random spanning trees. In *FOCS'09: Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science*, 2009.

[92] R. Khandekar, S. Rao, and U. Vazirani. Graph partitioning using single commodity flows. In *STOC'06: Proceedings of the 38th Annual ACM Symposium on Theory of Computing*, pages 385–390, New York, NY, USA, 2006. ACM.

[93] G. Kirchhoff. Uber die aufloesung der gleichungen auf welche man sei der untersuchung der linearen verteilung galvanischer strome gefuhrt wind. *Poggendorgs Ann. Phys. Chem.*, 72:497–508, 1847.

[94] P. Klein, S. Plotkin, C. Stein, and E. Tardos. Faster approximation algorithms for the unit capacity concurrent flow problem with applications to routing and finding sparse cuts. *SIAM Journal on Computing*, 23(3):466–487, 1994.

[95] P. Klein and N. Young. On the number of iterations for dantzig-wolfe optimization and packing-covering approximation algorithms. In *IPCO'02: Proceedings of the 7th International IPCO Conference*, pages 320–327. Springer, 2002.

[96] I. Koutis, G. L. Miller, and R. Peng. Approaching optimality for solving SDD systems. In *FOCS'10: Proceedings of the 51st Annual IEEE Symposium on Foundations of Computer Science*, 2010.

[97] V. G. Kulkarni. Generating random combinatorial objects. *Journal of Algorithms*, 11(2):185 – 207, 1990.

[98] E. Lawler, J. Lenstra, A. R. Kan, and D. Shmoys, editors. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. Wiley, New York, 1985.

[99] G. F. Lawler and A. D. Sokal. Bounds on the l^2 spectrum for markov chains and markov processes: A generalization of cheeger's inequality. *Transactions of the American Mathematical Society*, 309(2):577–580, 1988.

[100] T. Leighton, F. Makedon, S. Plotkin, C. Stein, E. Tardos, and S. Tragoudas. Fast approximation algorithms for multicommodity flow problems. *Journal of Computer and System Sciences*, 50(2):228–243, 1995.

[101] T. Leighton and S. Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM*, 46(6):787–832, 1999.

[102] N. Linial, E. London, and Y. Rabinovich. The geometry of graphs and some of its algorithmic applications. In *FOCS 1994: Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science*, pages 577–591, Washington, DC, USA, 1994. IEEE Computer Society.

[103] L. Lovász. Random walks on graphs: A survey. Combinatorics, Paul Erdös is eighty. Vol. 2 (Keszthely, 1993), 1993.

[104] R. Lyons and Y. Peres. *Probability on Trees and Networks*. 2009. In preparation. Current version available at http://mypage.iu.edu/~rdlyons/.

[105] A. Mądry. Fast approximation algorithms for cut-based problems in undirected graphs. In *FOCS'10: Proceedings of the 51st Annual IEEE Symposium on Foundations of Computer Science*, pages 245–254, 2010.

[106] A. Mądry. Faster approximation schemes for fractional multicommodity flow problems via dynamic graph algorithms. In *STOC'10: Proceedings of the 42nd Annual ACM Symposium on the Theory of Computing*, pages 121–130, 2010.

[107] H. Nagamochi and T. Ibaraki. Computing edge-connectivity in multigraphs and capacitated graphs. *SIAM Journal on Discrete Mathematics*, 5:54–66, February 1992.

[108] Y. Nesterov. Smooth minimization of non-smooth functions. *Mathematical Programming*, 103(1):127–152, 2005.

[109] Y. Nesterov. Fast gradient methods for network flow problems. In *The 20th International Symposium of Mathematical Programming*, 2009.

[110] L. Orecchia, L. J. Schulman, U. V. Vazirani, and N. K. Vishnoi. On partitioning graphs via single commodity flows. In *STOC'08: Proceedings of the 40th Annual ACM Symposium on the Theory of Computing*, pages 461–470, New York, NY, USA, 2008. ACM.

[111] A. Panconesi and A. Srinivasan. Randomized distributed edge coloring via an extension of the Chernoff-Hoeffding bounds. *SIAM Journal on Computing*, 26:350–368, 1997.

[112] C. Papadimitriou and S. Vempala. On the approximability of the traveling salesman problem. *Combinatorica*, 26:101–120, 2006. 10.1007/s00493-006-0008-z.

[113] S. A. Plotkin, D. B. Shmoys, and E. Tardos. Fast approximation algorithms for fractional packing and covering problems. *Mathematics of Operations Research*, 20:257–301, 1995.

[114] H. Räcke. Minimizing congestion in general networks. In *FOCS'02: Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science*, pages 43–52, Washington, DC, USA, 2002. IEEE Computer Society.

[115] H. Räcke. Optimal hierarchical decompositions for congestion minimization in networks. In *STOC'08: Proceedings of the 40th Annual ACM Symposium on the Theory of Computing*, pages 255–264, New York, NY, USA, 2008. ACM.

[116] T. Radzik. Fast deterministic approximation for the multicommodity flow problem. In *SODA'95: Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 486–492, Philadelphia, PA, USA, 1995. Society for Industrial and Applied Mathematics.

[117] L. Roditty and U. Zwick. Improved dynamic reachability algorithms for directed graphs. In *FOCS'02: Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science*, page 679, Washington, DC, USA, 2002. IEEE Computer Society.

[118] L. Roditty and U. Zwick. Dynamic approximate all-pairs shortest paths in undirected graphs. In *FOCS'04: Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science*, pages 499–508, Washington, DC, USA, 2004. IEEE Computer Society.

[119] L. Roditty and U. Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In *STOC'04: Proceedings of the 36th Annual ACM Symposium on the Theory of Computing*, pages 184–191, New York, NY, USA, 2004. ACM.

[120] L. Roditty and U. Zwick. On dynamic shortest paths problems. In *ESA'04: Proceedings of 12th Annual European Symposium on Algorithms*, pages 580–591, 2004.

[121] A. Schrijver. *Combinatorial Optimization*. Springer, 2003.

[122] F. Shahrokhi and D. W. Matula. The maximum concurrent flow problem. *Journal of the ACM*, 37(2):318–334, 1990.

[123] J. Sherman. Breaking the multicommodity flow barrier for $O(\sqrt{logn})$-approximations to sparsest cut. In *FOCS'09: Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science*, pages 363–372, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[124] D. B. Shmoys. Cut problems and their application to divide-and-conquer. In D. Hochbaum, editor, *Approximation Algorithms for NP-hard Problems*, pages 192–235, Boston, MA, USA, 1997. PWS Publishing Co.

[125] A. Sinclair and M. Jerrum. Approximate counting, uniform generation and rapidly mixing markov chains. *Information and Computation*, 82(1):93–133, 1989.

[126] D. A. Spielman. Algorithms, Graph Theory, and Linear Equations. In *Proceedings of International Congress of Mathematicians*, 2010.

[127] D. A. Spielman and N. Srivastava. Graph sparsification by effective resistances. In *STOC'08: Proceedings of the 40th Annual ACM Symposium on the Theory of Computing*, pages 563–568, 2008.

[128] D. A. Spielman and S.-H. Teng. Solving sparse, symmetric, diagonally-dominant linear systems in time $O(m^{1.31})$. In *FOCS'03: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, 2003.

[129] D. A. Spielman and S.-H. Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *STOC'04: Proceedings of the 36th Annual ACM Symposium on the Theory of Computing*, pages 81–90, New York, NY, USA, 2004. ACM.

[130] D. A. Spielman and S.-H. Teng. Nearly-linear time algorithms for preconditioning and solving symmetric, diagonally dominant linear systems. *CoRR*, abs/cs/0607105, 2006.

[131] D. A. Spielman and S.-H. Teng. A local clustering algorithm for massive graphs and its application to nearly-linear time graph partitioning. *CoRR*, abs/0809.3232, 2008.

[132] M. Stoer and F. Wagner. A simple min-cut algorithm. *Journal of the ACM*, 44:585–591, July 1997.

[133] M. Thorup and U. Zwick. Spanners and emulators with sublinear distance errors. In *SODA'06: Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 802–809, New York, NY, USA, 2006. ACM.

[134] L. Trevisan. Approximation algorithms for unique games. In *FOCS'05: Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science*, pages 05–34. IEEE Computer Society, 2005. Full version available at http://www.cs.berkeley.edu/ luca/pubs/unique.pdf.

[135] P. M. Vaidya. Solving linear equations with symmetric diagonally dominant matrices by constructing food preconditioners. Unpublished manuscript, UIUC 1990. A talk based on the manuscript was presented at the IMA Workshop on Graph Theory and Sparse Matrix Computation, October 1991, Mineapolis.

[136] N. T. Varopoulus. Isoperimetric inequalities and Markov chains. *Journal of Functional Analysis*, 63(2):215–239, 1985.

[137] S. Warshall. A theorem on Boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

[138] D. B. Wilson. Generating random spanning trees more quickly than the cover time. In *STOC'96: Proceedings of the 28th Annual ACM Symposium on the Theory of Computing*, pages 296–303. ACM, 1996.

[139] N. E. Young. Randomized rounding without solving the linear program. In *SODA'95: Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 170–178, Philadelphia, PA, USA, 1995. Society for Industrial and Applied Mathematics.

[140] U. Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM*, 49(3):289–317, 2002.