Exponential Start Time Clustering and its Applications in Spectral Graph Theory

Shen Chen Xu

CMU-CS-17-120
August 2017

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Gary L. Miller, Chair
Bernhard Haeupler
Daniel D. K. Sleator
Noel J. Walkington
Ioannis Koutis, University of Puerto Rico

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

Copyright © 2017 Shen Chen Xu

This research was sponsored by the National Science Foundation under grant numbers CCF-1065406, CCF-1637523, and CCF-1018463. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Spectral Graph Theory, Exponential Start Time Clustering, Graph Spanners, Spectral Graph Sparsification, Low Stretch Tree Embeddings, Hopsets

For my parents

Abstract

Recent progress on a number of combinatorial and numerical problems benefited from combining ideas and techniques from both fields to design faster and more powerful algorithms. A prime example is the field of spectral graph theory, which involves the interplay between combinatorial graph algorithms with numerical linear algebra. This led to the first nearly linear time solvers for graph Laplacians as well as symmetric and diagonally dominant (SDD) linear systems.

In this thesis we present several combinatorial algorithms that allow us to tap into spectral properties of graphs. In particular, we present

- An improved parallel algorithm for low diameter decomposition via exponential shifts.
- A parallel algorithm for graph spanners with near optimal stretch trade-offs and its application to spectral graph sparsification.
- Improved low stretch tree embeddings that are suitable for fast graph Laplacian solvers.
- Work efficient parallel algorithms for hopset and approximate shortest path.

A common goal we strive for in the design of these algorithms is to achieve complexities that are nearly linear in the input size in order to be scalable to the ever-growing amount of data and problem sizes in this day and age.

Acknowledgments

First and foremost, I would like thank my advisor Gary Miller for introducing me to the field of spectral graph theory, for his mentorship, constant support and being so generous with his time during my five years at Carnegie Mellon. I would also like to thank my thesis committee members Bernhard Haeupler, Ioannis Koutis, Daniel Sleator and Noel Walkington for their guidance and feedbacks during the dissertation process.

This thesis describes some results stemming from collaborations with Michael Cohen, Ioannis Koutis, Gary Miller, Jakub Pachocki, Richard Peng and Adrian Vladu. I also had the fortune to work with Kevin Deweese, John Gilbert, Michael Mitzenmacher, Charalampos Tsourakakis, Noel Walkington, Junxing Wang and Hao Ran Xu, as well as many thought provoking conversations with Hui Han Chin and Timothy Chu. I would like to thank Mark Wilde for being a mentor during my undergraduate study and getting me started on research. I also need to acknowledge all the work the staff of the Computer Science Department has put in to make our Ph.D. program such an awesome experience.

I am also grateful to my friends and fellow graduate students at Carnegie Mellon: Xiang Li, Jinliang Wei, Yuchen Wu, Yu Zhao, and many others for their friendships and support. Finally, I would like to thank my family for their love and support.

Contents

List of Figures xi

List of Algorithms xiii

1 Introduction 1

- 1.1 Preliminaries . . . 2
  - 1.1.1 Graphs and Their Laplacians . . . 2
  - 1.1.2 Graphs as Electrical Networks . . . 3
  - 1.1.3 Computational Goals . . . 4
- 1.2 Overview and Related Works . . . 5
  - 1.2.1 Fast Graph Laplacian Solvers . . . 5
  - 1.2.2 Spectral Graph Sparsification by Effective Resistances . . . 7
  - 1.2.3 Low Stretch Trees . . . 8
  - 1.2.4 Low Diameter Graph Decomposition . . . 10
  - 1.2.5 Hopsets . . . 10

2 Exponential Start Time Clustering 13

- 2.1 Low Diameter Graph Decomposition . . . 13
  - 2.1.1 The Exponential Shift . . . 14
- 2.2 The Clustering Algorithm . . . 15
- 2.3 Parallel Implementation on Unweighted Graphs . . . 20
  - 2.3.1 Remarks . . . 22

3 Parallel Graph Spanners and Combinatorial Sparsifiers 25

- 3.1 Introduction . . . 25
- 3.2 Spanners . . . 27
  - 3.2.1 The Unweighted Case . . . 27
  - 3.2.2 The Weighted Case . . . 29
- 3.3 Application to Combinatorial Sparsifiers . . . 31

4 Low Stretch Tree Embeddings 37

4.1 Introduction 37
4.1.1 Related Works 39
4.1.2 Applications 40
4.2 Embeddability 40
4.3 Overview of the Algorithm 42
4.4 Embeddable Trees from Bartal Decompositions 47
4.4.1 Bartal's Algorithm 47
4.4.2 Embeddability by Switching Moments 49
4.4.3 From Decompositions to Trees 52
4.5 Two-Stage Tree Construction 54
4.5.1 The AKPW Decomposition Routine 55
4.5.2 Accelerate Bartal's Algorithm using AKPW Decomposition 57
4.5.3 Decompositions that Ignore $1/k$ of the Edges 59
4.5.4 Bounding Expected ℓ_p-Stretch of Any Edge 62
4.5.5 Returning a Tree 66
4.6 Sufficiency of Embeddability 69

5 Parallel Shortest Paths and Hopsets **73**
5.1 Introduction 73
5.2 Hopset Construction 75
5.2.1 Hopsets in Unweighted Graphs 75
5.2.2 Hopsets in Weighted Graphs 80
5.3 Preprocessing of Weighted Graphs 82
5.4 Obtaining Lower Depth 84

6 Conclusions and Open Problems **89**

Bibliography **91**

List of Figures

2.1 Clustering generated by our algorithm on a 1000×1000 grid using different choices of βs. Different shades of gray represent different clusters 16

3.1 Known results on parallel algorithms for spanners, where $U = \frac{\max_e w(e)}{\min_e w(e)}$. 26

4.1 Bartal decomposition and the tree produced for a particular graph 46

5.1 Performances of Hopset Constructions, omitting ϵ dependency. 75

5.2 Interaction of a shortest path with the decomposition scheme. Hop set edges connecting the centers of large clusters allow us to "jump" from the first vertex in a large cluster (u), to the last vertex of a large cluster (v). The edges $\{u, c_1\}$, $\{c_2, v\}$ are star edges, while $\{c_1, c_2\}$ is a clique edge. 76

List of Algorithms

1 EST-Clustering(G, β) . . . 15
2 EST-Clustering-Impl(G, β) . . . 20
3 Unweighted-Spanner(G, k) . . . 27
4 Well-Separated-Spanner(G) . . . 30
5 Weighted-Spanner(G) . . . 30
6 Further-Sparsify$(G, \epsilon, \text{Spanner})$ . . . 33
7 Full-Sparsify$(G, \varepsilon, \text{Spanner})$ . . . 34
8 Decompose-Simple(G) . . . 48
9 Embeddable-Decompose(G, p, q) . . . 49
10 Build-Tree$(G, \mathbf{B})$ . . . 53
11 AKPW(G, δ) . . . 56
12 Decompose-Two-Stage$(G, \mathbf{d}, \mathbf{A})$ . . . 58
13 Decompose(G, p) . . . 66
14 Unweighted-Hopset(G, β) . . . 87

Chapter 1

Introduction

The past few years have seen significant developments of fast algorithms in areas such as symmetric and diagonally dominant (SDD) linear system solvers, numerical linear algebra, combinatorial optimization, and linear programming. An important idea underpinning this recent progress is to combine techniques developed in combinatorial and discrete algorithm design with numerical approaches. A prime example is the field of spectral graph theory, which involves the interplay between graph algorithms and linear algebra. For instance, the recent developments on nearly linear time solvers for the class of SDD linear systems [ST14, KMP14, KMP11, CKM+14] use discrete graph algorithms to find good preconditioners, while [KOSZ13], another nearly linear time solver, is entirely combinatorial. These solvers then lead to breakthroughs in long standing graph optimization problems such as finding maximum flows and shortest paths [CKM+11, LRS13, Mad13, KLOS14, CMSV16]. The ideas and techniques developed in the graph setting such as sampling and effective resistances/leverage scores also partly inspired a number of results in numerical linear algebra [CKM+11, LRS13, Mad13, KLOS14, CMSV16], and linear programming [LS14].

This thesis will focus on some of the combinatorial building blocks used in algorithmic spectral graph theory, as well as their applications. In Chapter 2 we describe a parallel low diameter graph decomposition routine which forms the basis for the next few chapters. In Chapter 3 we present parallel algorithms for finding graph spanners and its application to combinatorial constructions of spectral sparsifiers. In Chapter 4 we give efficient algorithms for a new type of low stretch tree embedding that plays an important role in solving graph Laplacians. In Chapter 5 we describe another application of our graph decomposition routine in parallel algorithms for approximating shortest paths.

1.1 Preliminaries

In this section we quickly introduce some of the notation and concepts used throughout this thesis.

1.1.1 Graphs and Their Laplacians

We use $G = (V, E, w)$ to denote an undirected weighted graph with vertex set V, edge set E, and non-negative edge weight function $w : E \to \mathbb{R}^+$. We will often use n to denote the number of vertices and m to denote the number of edges when there is no ambiguity about which graph we are referring to. It is also useful to define the reciprocal edge weight function as the edge length function $l : E \to \mathbb{R}^+$, $l(e) = 1/w(e)$ for all $e \in E$. Notice that we can also fully specify a graph using its length function as $G = (V, E, l)$. We then use $\mathrm{dist}(\cdot, \cdot)$ to denote the shortest path distance metric on G with respect to its edge length function l. We further assume that $w(u, v) = 0$ and $l(u, v) = \infty$ if u and v is not connected by an edge.

If $V' \subset V$, we use $G[V']$ to denote the induced subgraph of G on V'. In other words $G[V']$ is the graph with vertex set V' and edge set $E' = \{\{u, v\} \in E \mid u \in V', v \in V'\}$. Similarly if $E' \in E$, we use $G[E']$ to denote the induced subgraph with edge E' and vertex set $V' = \{v \in V \mid \exists u \in V, \{u, v\} \in E'\}$. Unless stated otherwise, we assume that induced subgraphs share the same edge weights of the original graph.

Given any $V' \subseteq V$, we can define $G \setminus V'$ to be the quotient graph of G obtained by contracting the connected components of $G[V']$. In other words, for $v \in V$, if $\textsc{Comp}(v)$ denotes the connected component of v in $G[V']$, then $G \setminus V'$ has vertex set $\{\textsc{Comp}(v) \mid v \in V\}$ and edge set $\{\{\textsc{Comp}(u), \textsc{Comp}(v)\} \mid \{u, v\} \in E\}$. We can similarly define quotient graphs $G \setminus E'$ and $G \setminus G'$ where $E' \subseteq E$ and G' is any subgraph of G. Again, the edge weights in these quotient graphs are assumed to be the same as in G unless otherwise stated.

Given an undirected weighted graph $G = (V, E, w)$, its weighted adjacency matrix $\mathbf{A}_G$ is defined to the symmetric matrix with rows and columns indexed by the vertex set V, with the off-diagonal entries given by the edge weights:

$$(\mathbf{A}_G)_{u,v} = w(u, v).$$

The diagonal degree matrix $\mathbf{D}_G$ is defined as

$$(\mathbf{D}_G)_{u,u} = \sum_{v \neq u} w(u, v).$$

The graph Laplacian L_G associated with G is then given by

$$\mathbf{L}_G = \mathbf{D}_G - \mathbf{A}_G.$$

An alternative way to characterize graph Laplacians is via the vertex-edge incidence matrix. First, we arbitrarily orient the edges of G, i.e. we write each edge $(u, v) \in E$ as an ordered pair. We define $\mathbf{B}_G$ to be the $|V| \times |E|$ matrix whose rows are indexed by vertices and whose columns are indexed by edges:

$$(\mathbf{B}_G)_{v,e} = \begin{cases} 1 & \text{if } e = (v, u) \text{ for some } u \in V, \\ -1 & \text{if } e = (u, v) \text{ for some } u \in V, \\ 0 & \text{otherwise.} \end{cases}$$

We let $\mathbf{C}_G$ be the $|E| \times |E|$ diagonal matrix of edge weights

$$(\mathbf{C}_G)_{e,e} = w(e).$$

Then it is easy to verify that

$$\mathbf{L}_G = \mathbf{B}_G \mathbf{C}_G \mathbf{B}_G^T.$$

This characterization also shows that Laplacians are positive semi-definite matrices when the corresponding graphs have non-negative edge weights.

If necessary, all the above notations will be augmented with superscripts or subscripts to describe further restrictions or disambiguation, which we will make explicit when the occasion arises.

1.1.2 Graphs as Electrical Networks

It is often useful to view graphs and Laplacians as a model of electrical circuits. Here we highlight some concepts that will be useful to us (a detailed treatment of this topic can be found in [DS84]).

Under this view of the world, given a graph $G = (V, E, l)$, the vertex set V represents the set of junctions in the circuit, and the edge set E are the resistors, whose resistances are given by the length function $l : E \to \mathbb{R}^+$ (and therefore the conductances are given by the edge weight function $w(e) = 1/l(e)$). We will first choose an arbitrary orientation for each edge just as before. The graph itself is still undirected, but since in our model an edge can carry electric current in either direction, choosing an orientation allows us to describe the current using a single real number: a positive current is in the same direction as the edge orientation, a negative current goes in the opposite direction.

Then given $x, b \in \mathbb{R}^{|V|}$, we can interpret the system of linear equations $\mathbf{L}x = b$ as follows. We will view the entries of x as electric voltage values at each vertex. Then since $\mathbf{L}_G = \mathbf{B}_G \mathbf{C}_G \mathbf{B}_G^T$,

applying $\mathbf{L}_G$ to x we can write

$$\begin{aligned}\mathbf{L}_G x &= \mathbf{B}_G \mathbf{C}_G \mathbf{B}_G^T x \\ &= \mathbf{B}_G \mathbf{C}_G \delta \\ &= \mathbf{B}_G f \\ &= b,\end{aligned}$$

where $\delta, f \in \mathbb{R}^{|E|}$. Using Ohm's law and the definition of the vertex-edge incidence matrix $\mathbf{B}_G$, we see that $\delta \in \mathbb{R}^{|E|}$ represents the voltage difference between the endpoints of each edge, and $f \in \mathbb{R}^{|E|}$ represents electric current on each edge (with the signs determined by the edge orientation). Similarly, the vector $b \in \mathbb{R}^{|V|}$ represents the net amount of electric current entering/leaving at each node, if we set the voltages according to x.

An important concept is the effective resistances between two vertices. Informally, for any $u, v \in V$, the effective resistance $\mathrm{ER}(u, v)$ between them can be measured by applying Ohm's law to u and v with the rest of the network viewed as a single resistor. That is, it is equal to the potential difference between u and v when we inject one unit of electric current into u and extract the same amount out of v. To formally define the effective resistances, let $b_{u,v}$ be the vector with 1 in the entry corresponding to u, -1 in the entry corresponding to v and 0 everywhere else. Then using $b_{u,v}$ as the right hand side, the solution to $\mathbf{L}_G x = b_{u,v}$ then gives the voltage values that would induce one unit of electric current from u to v, and therefore we have

$$\mathrm{ER}(u, v) = b_{u,v}^T \mathbf{L}_G^{-1} b_{u,v}.$$

As one would expect, the effective resistances can only increase when we increase the resistances on the individual edges, and this is known as Rayleigh's monotonicity principle.

Lemma 1.1.1 (Rayleigh's Monotonicity Principle)**.** *If $H = (V, E_H, w_H)$ is a subgraph of $G = (V, E_G, w_G)$ in the weighted sense, i.e. $E_H \subseteq E_G$ and $w_H(e) \leq w_G(e)$ for all $e \in E_H$, then for any $u, v \in V$,*

$$\mathrm{ER}_H(u, v) \geq \mathrm{ER}_G(u, v).$$

1.1.3 Computational Goals

With the rise of the Internet and the age of big data, we saw an explosive growth in the demand for information processing tasks such as data and network analysis. During this day and age it is essential for our algorithms to be scalable to massive problem sizes. For example an algorithm that runs in $O(n^2)$ time may no longer be suitable for solving today's problem, and will surely not scale up to the problems of the near future. Therefore we will focus on

algorithms with complexities that are *nearly linear* in the input size. In other words, on a problem of size n, we would like to design algorithms that runs in $O(n \log^c n)$ time for some constant c.

We will also be interested in the design of parallel algorithms. We will use two standard quantities to measure the complexity of a parallel algorithm: *depth* and *work*. The depth (D) of a parallel algorithm is the length of the longest sequential dependencies of the algorithm, i.e. a sequence of computations in which latter ones depends on the result of earlier ones. Work (W) on the other hand is defined the total number of operations performed by the algorithm. In practice, the number of processors available (P) is often limited, therefore if $W/P > D$, the actual running time of the algorithm no longer depends on the depth, but is rather bottlenecked on the total work divided by the number of processors. Therefore in this thesis, we focus on designing *work efficient* parallel algorithms. In other words, we want to parallel algorithms whose work term match the sequential run time as closely as possible (up to a poly-logarithmic factor), since these algorithms are able to achieve parallel speedup with only a modest number of processors.

1.2 Overview and Related Works

1.2.1 Fast Graph Laplacian Solvers

Many recent developments in algorithmic spectral graph theory were started by a series of ground breaking papers by Spielman and Teng [ST03, ST11, ST13, ST14] that resulted in the first nearly linear time solver for linear systems in graph Laplacians (and as an extension, for symmetric and diagonally dominant systems). On a graph with n vertices and m edges, the running time[1] for their algorithm was $\tilde{O}(m \log^c n)$[2] for some fairly big constant c. Since then this running time has been improved, first to $\tilde{O}(m \log^2 n)$ [KMP14], then to $\tilde{O}(m \log n)$ [KMP11], and finally to $\tilde{O}(m\sqrt{\log n})$ [CKM+14]. At the core of these solvers is an iterative and recursive scheme for solving linear systems combined with combinatorial algorithms for finding good preconditioners.

Iterative methods for solving linear systems are often used to solve linear systems that are large and sparse (we refer readers to [Saa03] for an introduction on iterative methods). One drawback of direct methods such as LU factorization (also known as Cholesky factorization for symmetric and positive semi-definite matrices such as graph Laplacians) is that they can produce large amounts of *fill-in*. That is, during factorization, many entries of the matrix that were initially zero can become non-zero. For an $n \times n$ matrix with only $O(n)$ non-zero entries,

[1]The solvers presented in this section all produce an approximate solution to the linear system, but for simplicity we ignore the running time dependency on the error parameter ϵ, which it is typically an extra factor of $O(\log(1/\epsilon))$.

[2]We use $\tilde{O}(\cdot)$ to ignore $O(\text{poly} \log \log n)$ factors in addition to constant factors in this section.

computing its LU factors can easily produce $\Omega(n^2)$ non-zero entries. On a large sparse matrix, this not only means a super-linear running time, but it also requires what is often an impossibly large amount of memory.

Iterative methods, on the other hand, produce a sequence of improving approximate solutions with very little memory overhead. As an example let us consider one of the simplest example of iterative solvers, the Richardson iteration. Given a linear system $\mathbf{A}x = b$ and an initial guess $x^{(0)}$, this method tries to improve the current solution using the following update rule:

$$x^{(k+1)} = x^{(k)} + (b - \mathbf{A}x^{(k)}).$$

Each iteration consists of a matrix-vector multiplication and vector additions, thus has a linear running time with no memory overhead. However, the convergence of this iterative method (if it converges at all) depends on the condition number of the matrix $\mathbf{A}$, which is defined to be the ratio between the largest and the smallest eigenvalues of $\mathbf{A}$. If $\mathbf{A}$ is ill-conditioned, iterative methods can be painfully slow.

One way to improve the convergence of iterative methods is via preconditioning. In place of the original system, we introduce a preconditioner matrix $\mathbf{P}$, chosen so that $\mathbf{P}^{-1}\mathbf{A}$ has a good condition number, and try to solve the equivalent system $\mathbf{P}^{-1}\mathbf{A}x = \mathbf{P}^{-1}b$ instead. It can be shown that the condition number of $\mathbf{P}^{-1}\mathbf{A}$ is bounded by β/α if

$$\alpha x^T\mathbf{P}x \leq x^T\mathbf{A}x \leq \beta x^T\mathbf{P}x, \quad \forall x \in \mathbb{R}^n. \tag{1.1}$$

That is, we want the quadratic form of the preconditioner to approximate that of the original matrix. The preconditioned Richardson iteration then becomes

$$\begin{aligned} x^{(k+1)} &= x^{(k)} + (\mathbf{P}^{-1}b - \mathbf{P}^{-1}\mathbf{A}x^{(k)}) \\ &= x^{(k)} + \mathbf{P}^{-1}(b - \mathbf{A}x^{(k)}). \end{aligned}$$

Since each iteration now requires an application of $\mathbf{P}^{-1}$, in other words a solution to the linear system in $\mathbf{P}$, the preconditioner should also be under some notion easy to solve, either directly, or in the framework of Spielman and Teng, recursively.

Although it is not known how to find provably good preconditioner for general matrices, we do have algorithms for finding preconditioners of graph Laplacians that gives us nearly linear time solvers. An important idea, which can be traced back to Vaidya [Vai91], is to use another graph as the preconditioner and to leverage combinatorial graph algorithms to find it. Given this, the goal is then to find this new graph such that it approximates the original graph in the sense of (1.1). It can be shown that for graph Laplacians

$$x^T\mathbf{L}x = \sum_{\{u,v\}\in E} w(u,v)(x_u - x_v)^2.$$

In other words, the quadratic form of graph Laplacians can be interpreted as the electric energy dissipated under the voltage setting x when we view the graph a resistive network. Spielman and Srivastava [SS11] showed that the effective resistances of the edges are crucial to construct sparser graphs, also known as spectral sparsifiers, that approximate the quadratic form of the original Laplacians. The series of work [KMP14, KMP11] that lead to the state of art solver [CKM+14] all construct their preconditioners by sampling edges using upper bounds of effective resistances to form a spectral sparsifier that is at the same time also easier to recursively solve. These upper bounds are obtained by first finding a low stretch spanning tree of the graph, then applying Rayleigh's monotonicity principle (Lemma 1.1.1) to the effective resistances in the tree.

The first part of this thesis can be thought of as combinatorial algorithms used in obtaining these relatively crude effective resistance upper bounds used in preconditioners (Chapter 4) as well as tighter estimates of effective resistances suitable for more accurate spectral sparsification (Chapter 3).

Aside from the recursive preconditioning framework pioneered by Spielman and Teng, a different nearly linear time solver for graph Laplacians was given by Kelner *et al.* [KOSZ13]. Recall from Section 1.1.2 that solving the system $\mathbf{L}x = b$ can be interpreted as finding a voltage setting x on each vertex that will induce an electric current the satisfies the demand at each vertex given by b. Instead of the correct voltage values, the solver by Kelner *et al.* tries to solve the dual problem of directly computing the electric current induced by these voltages. The run time of their algorithm is $\tilde{O}(m \log^2 n)$ and this was further improved by Lee and Sidford [LS13] to $\tilde{O}(m \log^{3/2} n)$.

1.2.2 Spectral Graph Sparsification by Effective Resistances

In the previous section we saw that in order to improve the convergence of an iterative method, a preconditioner need to "spectrally" approximate the original system. In this section we give an overview on graph sparsification, the problem of finding sparsest possible graphs that spectrally approximate the original. Although being spectrally similar is a necessary but not sufficient property for being a good preconditioner, graph sparsification is an interesting graph problem in itself. Given a graph $G = (V, E, w)$ and its Laplacian $\mathbf{L}_G$ and any given error parameter $0 < \epsilon < 1$, our goal is to find a sparse graph H such that

$$(1-\epsilon)x^T\mathbf{L}_G x \leq x^T\mathbf{L}_H x \leq (1+\epsilon)x^T\mathbf{L}_G x, \qquad \forall x \in \mathbb{R}^n.$$

We will call such a graph H a $(1 \pm \epsilon)$-*spectral sparsifier* of G.

Spectral sparsification of graphs was introduced by Spielman and Teng as a component in the first nearly-linear time SDD linear system solver [ST11]. Their sparsification algorithm is combinatorial in nature, it relies on intricate graph partitioning followed by uniform sampling

in some of the partitions. However this produces a sparsifier of size $O(\frac{n \log^c n}{\epsilon^2})$ for a fairly large constant c. Spielman and Srivastava [SS11] later introduced an elegant sparsification algorithm based on sampling edges by effective resistances. More specifically, the sampling bias p_e for an edge e is formed by scaling its effective resistance by the edge weight:

$$p_e = w(e)\mathrm{ER}(e). \tag{1.2}$$

This sampling scheme combined with Chernoff-type bounds for positive semi-definite matrices [Tro12] gives $(1 \pm \epsilon)$-spectral sparsifiers with at most $O(\frac{n \log n}{\epsilon^2})$ edges for any input graph. The bound on the sparsifier size was further improved to $O(\frac{n}{\epsilon^2})$ by [BSS12, LS17] using different techniques.

However these recent sparsification algorithms require multiple solutions to linear systems in graph Laplacians or semi-definite programs. In particular, the algorithm from [SS11] requires about $O(\frac{\log n}{\epsilon^2})$ graph Laplacian solves to compute the effective resistances of all the edges to sufficient accuracy. Recent efforts have been made in trying to design better *combinatorial* algorithms for graph sparsification, which is more desirable from a practical standpoint.

Koutis, Miller and Peng [Kou14] showed that the sampling biases $\{p_e\}_{e \in E}$ from [SS11] can be replaced by any set of values $\{u_e\}_{e \in E}$ as long as $u_e \geq p_e$ for all $e \in E$. Then if $\sum_{e \in E} u_e = U$, the same sampling scheme yields a $(1 + \epsilon)$-sparsifier with about $O(\frac{U \log U}{\epsilon^2})$ edges (in particular, the original result by Spielman and Srivastava uses a theorem by Foster [Fos49] which states $\sum_{e \in E} p_e = n - 1$). This allows us to use combinatorial means to estimate effective resistances while trading off on the size of the resulting sparsifier. In [Kou14] Koutis showed how to compute estimates of effective resistances by finding a bundle of disjoint graph spanners, and gave a combinatorial and parallel algorithm for constructing sparsifiers of size $O(\frac{n \log^2 n \log^2 \rho}{\epsilon^2} + \frac{m}{\rho})$ for any parameter ρ. In Chapter 3, we present improved parallel algorithms for spanners and tighten the argument from [Kou14] to obtain sparsifiers with at most $O(\frac{n \log^2 n}{\epsilon^2})$ edges using only combinatorial means.

1.2.3 Low Stretch Trees

Recall that a good preconditioner tries to strike a balance between being a good spectral approximation of the original system and being under some notion easier to solve. The graph sparsifiers we saw in the previous section, while being very good spectral approximations and sparse in edge count, are not necessarily easier to solve than dense graphs. It turns out that, just like many other graph problems, the easiest Laplacians to solve using direct methods are those corresponding to trees, as they generate zero fill-in during elimination. The first graph preconditioning algorithm by Vaidya [Vai91] uses a maximum weight spanning tree augmented with some off-tree edges. Boman and Hendrickson [BH01] later pointed out a different class

of trees known as low stretch spanning trees that are more suitable for preconditioning graph Laplacians. Recent works on nearly linear time solvers [KMP14, KMP11, CKM+14] all employ a low stretch tree augmented with some off-tree edges as their preconditioners.

Let G be a weighted graph and T a spanning tree of G. For each edge $e = \{u, v\}$ in G, the stretch of e with respect to T is defined as

$$\begin{aligned} \text{str}_T(e) &\stackrel{\text{def}}{=} \frac{\text{dist}_T(u,v)}{l(e)} \\ &= w(e) \cdot \text{dist}_T(u,v), \end{aligned}$$

where $\text{dist}_T(\cdot,\cdot)$ is the shortest path distance in T and $l(e)$ is the length of e. Since T is a subgraph of G, by Rayleigh's monotonicity law effective resistances in G are upper bounded by those in T. Since T is a tree, effective resistances in T are nothing more than the shortest path distances and are trivial to compute. Recall that under our definition edge weights and edge lengths are reciprocals of each other, thus the stretch of an edge is in fact an upper bound on the sampling bias of the edge in the Spielman-Srivastava approach to spectral sparsification (see Equation (1.2)). It is then natural to ask for a tree that minimizes the sum of stretches of all the edges.

Alon *et al.* [AKPW95] introduced the notion of the low stretch spanning tree and gave an algorithm for constructing spanner trees with an average stretch per edge of $\exp(O(\sqrt{\log n \log\log n}))$. This was subsequently improved by Elkin *et al.* [EEST08] to $O(\log^2 n \log\log n)$, then by Abraham *et al.* [ABN08] to $O(\log n \log\log n (\log\log\log n)^3)$. More recently Abraham and Neiman [AN12] showed how to construct spanning trees with $O(\log n \log\log n)$ average stretch, approaching the optimal and conjectured[3] bound of $O(\log n)$. Their algorithm runs in $O(m \log n \log\log n)$ time and is used in the $O(m \log n \log\log n)$ time solver [4] by Koutis *et al.* [KMP11].

Generally speaking, trees with lower average stretch will lead to faster solvers for the graph Laplacians under the framework pioneered by Spielman and Teng [ST14] and subsequently improved by [KMP14, KMP11, CKM+14]. In order to obtain the $\tilde{O}(m\sqrt{\log n})$ time Laplacian solver in [CKM+14], we introduce two relaxations to the low stretch spanning tree objective.

First we relax the requirement for the tree to be a spanning tree, and only ask the tree to be *embeddable* into the original graph. In other words, we require a mapping from edges in T to paths in G such that under this map T is a subgraph of G. This is closely related to the problem approximating arbitrary metrics with tree metrics. For this problem Bartal [Bar98] first gave an algorithm with $O(\log n \log\log n)$ expected stretch and Fakcharoenphol *et al.* [FRT04] subsequently improved this bound to the optimal $O(\log n)$ expected stretch. We further introduce

[3] By Alon *et al.* [AKPW95].

[4] All the solver runtimes in this section omit a $O(\log(1/\epsilon))$ factor.

the notion of ℓ_p-stretch: for $p < 1$, we let

$$\text{str}_T^p(e) \stackrel{\text{def}}{=} (\text{str}_T(e))^p.$$

Compared to $p = 1$, this relaxation allows us to discount the cost of highly stretched edges, giving an average ℓ_p stretch of $O((\frac{1}{1-p})^2 \log^p n)$, but is still suitable for the iterative methods used in the solver.

The other bottleneck to a faster solver is the runtime of the tree finding algorithms, as state-of-the-art algorithms for low stretch spanning trees runs in time around $O(m \log n)$. Here we relax the restriction of only using spanning trees, and consider Steiner trees (trees with extra vertices) as long as they can be embedded into the original graph. To this end we combine the bottom-up algorithm of Alon *et al.* [AKPW95] (a linear time algorithm with relatively poor stretch guarantee) with the top down approach of Bartal [Bar98] (an expensive algorithm with good stretch guarantee) to obtain a $O(m \log \log n)$ time algorithm. Combining these two relaxations we obtain an improved tree embedding for Laplacian solvers, which is described in detail in Chapter 4.

1.2.4 Low Diameter Graph Decomposition

We just saw the important role of effective resistances in spectral graph algorithms. One can show that effective resistances in graphs in fact form a metric on the set of vertices, and most of the algorithms presented in this thesis can be viewed as combinatorially approximating this metric. The basic building block we are going to employ is the familiar shortest path metric on graphs and low diameter decomposition. Originally introduced by Awerbuch [Awe85], the low diameter decomposition aims to partition a graph into pieces such that distances within each piece are small and few edges span different pieces. In Chapter 2 we present a randomized parallel algorithm for computing these decompositions with optimal parameters. This will serve as an important building block for rest of this thesis.

1.2.5 Hopsets

In addition to finding low stretch trees and spanners for spectral sparsification, our low diameter graph decomposition algorithm can also be applied to approximating shortest paths in parallel. When the edge lengths are non-negative, the shortest path problem has a $O(\text{poly} \log n)$ depth parallel algorithm based on repeated squaring of the adjacency matrix. This algorithm however incurs $O(n^3)$ work, significantly more than the sequential algorithm by Dijkstra [Dij59]. Unfortunately work efficient parallel algorithms for exact shortest paths remained elusive, thus researchers have turned to approximations instead. Most of these approximation results use a construct known as hopset, a term coined by Cohen [Coh00] but the concept itself

appeared in several earlier works [UY91, KS97]. A hopset is an extra set of edges which, when added to the graph, guarantees that any shortest path can be approximated by a path few edges.

The bottleneck that prevents work efficient algorithms such as breadth first search to become parallel is the fact that an exact shortest paths can contain up to $O(n)$ edges. To find such a path, breadth first search needs to at least traverse that path, resulting in $O(n)$ depth. Thus if we are willing to settle with approximation, this bottleneck can be circumvented as we can first compute a hopset and then apply breadth first search to the augmented graph. In Chapter 5 we will give constructions of hopsets that lead to parallel algorithms for approximating shortest paths. Compared to previous results [UY91, KS97, Coh00], our algorithm is the first to achieve sub-linear depth and $O(m \operatorname{poly} \log n)$ work.

Chapter 2

Exponential Start Time Clustering

2.1 Low Diameter Graph Decomposition

Low diameter decomposition is the problem of partitioning a graph into clusters with small diameter, such that few edges have their endpoints in two different clusters. We notice that these are two conflicting objectives: At one extreme we can achieve minimal diameter by putting each vertex into its own cluster but as a result each edge spans two clusters; On the other hand we can leave the entire graph as a single cluster. This can result in a diameter of up to $O(n)$, but no edges are cut.

Low diameter decomposition is a fundamental algorithmic tool in spectral graph theory. It forms the basis of algorithms for low stretch spanning trees [AKPW95, EEST08] and low stretch tree embeddings [Bar98, FRT04, CMP+14], which play a crucial role in fast graph Laplacian solvers [ST14, KMP14, KMP11, CKM+14]. It also has applications in distributed computing [Awe85, EN16], graph spanners [PS89, MPVX15], spectral sparsifiers [KP12, Kou14], and various other graph optimization problems [CKR05, FHRT03, MPVX15].

Given a subset of vertices $S \subseteq V$, its diameter can be defined in two ways. The *strong* diameter is the maximum distance between two vertices in the induced subgraph on S. The *weak* diameter, on the other hand, is the maximum distance between two vertices in S, where the distance is measured in the original graph (i.e. the shortest path go can outside of S). In this thesis, we will work with the stronger notion of cluster diameter, as it is crucial for us to certify distances using a spanning tree within a cluster. In particular, we use the following probabilistic definition of low diameter graph decomposition.

Definition 2.1.1. *Given a possibly weighted graph $G = (V, E, l)$ with vertex set V, edge set E and edge lengths $l : E \to \mathbb{R}^+$, a (β, d)-decomposition of G is distribution over partitions of the $V(G)$ into clusters $\{C_1, C_2, \ldots, C_k\}$ such that*

1. *The strong diameter of each C_i is at most d with high probability.*
2. *For each edge $e \in E$, the endpoints of e are in different clusters with probability at most $\beta l(e)$.*

Awerbuch [Awe85] first introduced the above low diameter decomposition problem and gave a sequential and deterministic algorithm for decompositions with $O(\frac{\log n}{\beta})$ diameter cutting at most a β fraction of the edges. The algorithm is very simple to describe on unweighted graphs: clusters are sequentially formed by choosing an arbitrary starting vertex and performing a breadth first search until the number of outgoing edges are at most a β fraction of the intern edges. Bartal [Bar96, Bar98] later gave a randomized construction for $(\beta, O(\frac{logn}{\beta}))$-decompositions as defined above, where the radius of the BFS generated clusters is chosen uniformly at random. Both of these algorithms can be thought of as repeated ball growing: starting at an arbitrary vertex, a cluster is formed by including all vertices within a certain distance, chosen in a way to balance the diameter and the number of edges cut. The cluster is then removed and this procedure repeats until the graph is exhausted.

In their development of parallel SDD linear system solver, Blelloch *et al.* [BGK+14] gave a parallel ball growing construction, producing a $(\beta, O(\frac{\log^2 n}{\beta}))$-decomposition. A main difficulty in parallelizing the ball growing algorithm is in controlling work spent examining the same parts of the graph as different clusters (growing in parallel) collide and overlap. Since the number of pieces in the final decomposition maybe large (e.g. on the line graph), any parallel algorithm must be at some point constructing a large number of pieces simultaneously. On the other hand, for highly connected graph such as the expander graphs, growing too many clusters in the same time can result in large amount of overlaps between clusters and total work quadratic in the size of the graph. Additionally, how to resolve these overlaps in such way that few edges are cut is also a non-trivial task. The method given in [BGK+14] is to geometrically increase the number of parallel ball growing and introduce random backoffs when two or more clusters collide. In this chapter we present a simple and streamlined parallel and distributed algorithm for finding $(\beta, O(\frac{\log n}{\beta}))$-decompositions, which will become an important building block for the next few chapters. This algorithm is based on exponential shifts of start times in the parallel graph search and first appeared in [MPX13, MPVX15].

2.1.1 The Exponential Shift

In order to obtain parallel ball growing algorithm with comparable guarantees as the sequential counterpart, we need satisfactory answers to the following two questions: how many and which clusters should we be growing at any given time, and what to do when different clusters collide. To answer the first question, we introduce a random shift in the start time of each individual ball. When the boundaries of two clusters collide, we will simply have them stop expanding at that point, thus introducing zero overlaps. This random shift in the start time will come from the exponential distribution.

The exponential distribution is a continuous probability distribution parameterized by a *rate parameter* λ. We use $X \sim \text{Exp}(\lambda)$ to denote that X is exponentially distribution with parameter λ. Its probability density function is given by

$$f_X(x;\lambda) = \begin{cases} \lambda \exp(-\lambda x) & \text{if } x \geq 0, \\ 0 & \text{otherwise.} \end{cases}$$

and its cumulative distribution function is given by

$$F_X(x;\lambda) = \begin{cases} 1 - \exp(-\lambda x) & \text{if } x \geq 0, \\ 0 & \text{otherwise.} \end{cases}$$

The exponential distribution has an important property known as the *memoryless property*, and the correctness of our algorithm relies on this fact.

Fact 2.1.2. *If X is an exponential random variable, then we have that*

$$\mathbf{Pr}\left[X > s + t \mid X > s\right] = \mathbf{Pr}\left[X > t\right], \quad \forall s, t \geq 0.$$

2.2 The Clustering Algorithm

Our clustering algorithm is given in Algorithm 1. Figure 2.1 shows the result clustering of a 1000×1000 square grid graph with different choices of β. As we expected, smaller β leads to larger diameter pieces and fewer edges on the boundaries.

Algorithm 1 EST-CLUSTERING(G, β)

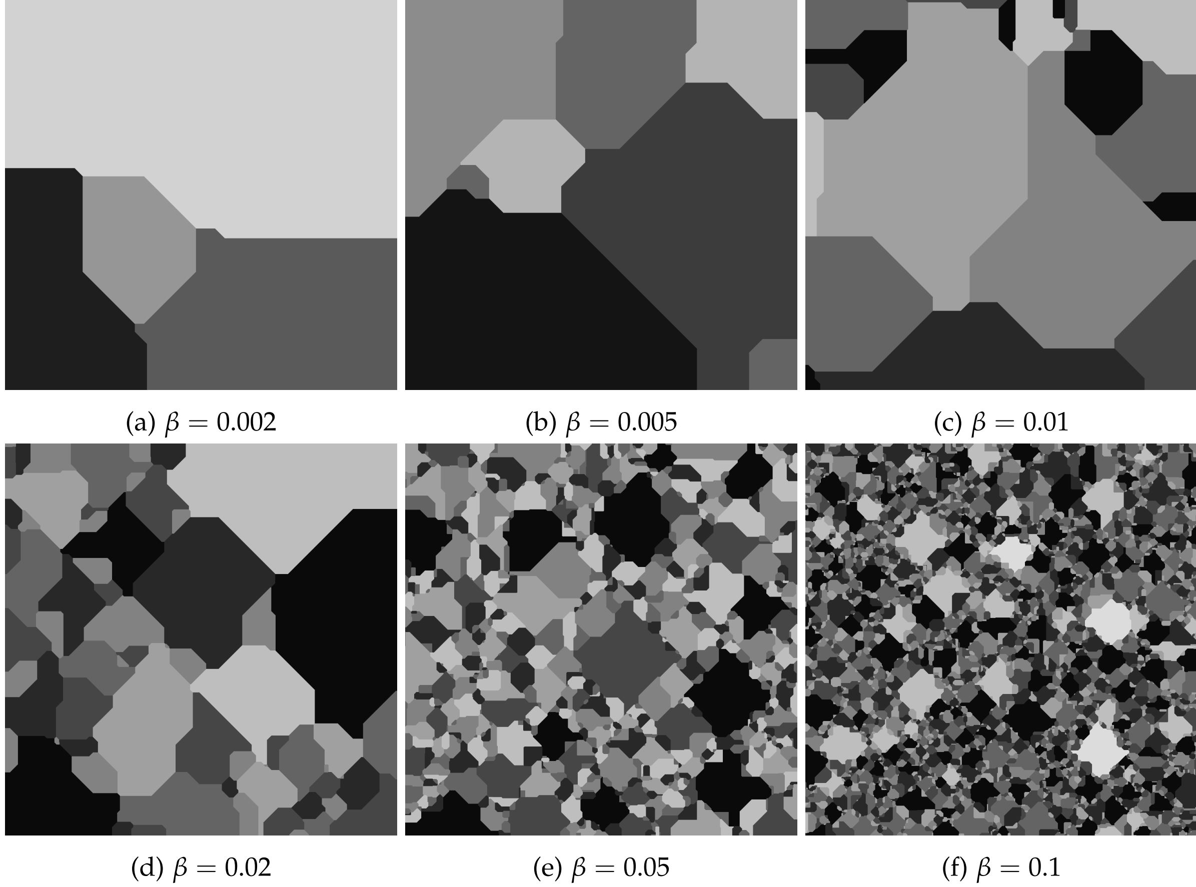
Input: Weighted graph $G = (V, E, l)$ and parameter β.
Output: A $(\beta, O(\frac{\log n}{\beta}))$-decomposition of G with high probability

1: draw independent random variables $\delta_v \sim \text{Exp}(\beta)$ for each $v \in V$
2: $c(v) \leftarrow \arg\min_u \text{dist}(u, v) - \delta_u$ for each $v \in V$, breaking ties lexicographically
3: $C_v \leftarrow \{u \in V \mid c(u) = v\}$ for each $v \in V$
4: **return** $\{C_v \mid C_v \neq \varnothing\}$

Formally, each vertex is assigned to the cluster whose center is the closest according to a exponentially shifted distance. One can also think of this algorithm as parallel ball growing where vertices are given a random "head start" drawn from an exponential distribution. This interpretation will be used when we discuss the implementation of this algorithm in Section 2.3. For now, we concern ourselves with the correctness of this algorithm. For the rest of this section define this shifted distance

$$\text{dist}_\delta(u, v) \stackrel{\text{def}}{=} \text{dist}(u, v) - \delta_u.$$

Notice that $\text{dist}_\delta(\cdot, \cdot)$ is not symmetric.



(a) $\beta = 0.002$ (b) $\beta = 0.005$ (c) $\beta = 0.01$

(d) $\beta = 0.02$ (e) $\beta = 0.05$ (f) $\beta = 0.1$

Figure 2.1: Clustering generated by our algorithm on a 1000×1000 grid using different choices of βs. Different shades of gray represent different clusters

Lemma 2.2.1. *If* EST-CLUSTERING *assigns v to the cluster C_u centered at u, then the next vertex v' on the shortest path from v to u is also assigned to C_u.*

Proof. We give a proof by contradiction. Suppose $v' \in C_{u'}$ for some $u' \neq u$. Notice that $\text{dist}_\delta(u, v') = \text{dist}_\delta(u, v) + 1$ and $\text{dist}_\delta(u', v) \leq \text{dist}_\delta(u', v') + 1$. Since EST-CLUSTERING assigned v' to u' instead of u, it must be one of the following two cases:

1. We have $\text{dist}_\delta(u', v') < \text{dist}_\delta(u, v')$. Combined with the above observations we have

$$\begin{aligned} \text{dist}_\delta(u', v) &\leq \text{dist}_\delta(u', v') + 1 \\ &< \text{dist}_\delta(u, v') + 1 \\ &= \text{dist}_\delta(u, v). \end{aligned}$$

   This is a contradiction since v is strictly closer to u' in terms of the shifted distance and should have been assigned to $C_{u'}$.

2. We have $\text{dist}_\delta(u'v') = \text{dist}_\delta(u, v')$. A similar argument gives that $\text{dist}_\delta(u', v) \leq \text{dist}_\delta(u, v)$. Since we broke the ties lexicographically, it must be case that u is also assigned to v'.

■

Notice that the second case in the proof above is actually a zero probability event. However this is useful for when we discuss the implementation of this algorithm where numbers will be rounded to fixed precision.

Lemma 2.2.2. *The strong diameter of each cluster is $O(\frac{\log n}{\beta})$ with high probability.*

Proof. Using Lemma 2.2.1, it suffices to bound the distance from any vertex to its center (i.e. the radius of each cluster around its center). If the vertex v is assigned to the center u, we must have

$$\begin{aligned} \text{dist}_\delta(u, v) &\leq \text{dist}_\delta(v, v) \\ \text{dist}(u, v) - \delta_u &\leq \text{dist}(v, v) - \delta_v \\ &= -\delta_v \\ &\leq 0. \end{aligned}$$

The last inequality follows from the fact that exponential random variables are non-negative, and this gives

$$\text{dist}(u, v) \leq \delta_u.$$

Therefore $\max_{v \in V} \delta_v$ is an upper bound on the radius of each piece. Using the cumulative distribution function of the exponential distribution, for any constant $k > 0$ we have

$$\begin{aligned} \mathbf{Pr}\left[\delta_v > (k+1)\frac{\log n}{\beta}\right] &= \exp\left(-\beta(k+1)\frac{\log n}{\beta}\right) \\ &= n^{-(k+1)}. \end{aligned}$$

Applying union bound over all the n vertices then gives us an overall failure probability of n^{-k}.

■

We now analyze the probability of EST-Clustering shattering a fixed subgraph into multiple clusters. Define $B(c, r) = \{v \in V \mid \text{dist}(c, v) \leq r\}$ to be the ball of radius r centered around c.

Lemma 2.2.3. *For any fixed point c, radius $r > 0$ and $k \geq 1$, the probability that $B(c, r)$ intersects k or more clusters in the output of* EST-Clustering *is at most* $(1 - \exp(-2r\beta))^{k-1}$.

Proof. The case where $k = 1$ is trivial. Suppose u and v are two of the centers whose clusters intersect with $B(c, r)$, and without loss of generality suppose $\text{dist}_\delta(u, c) \leq \text{dist}_\delta(v, c)$. Let

$$u' = \arg\min_{w \in B(c,r)} \text{dist}(u, w)$$

and

$$v' = \arg\min_{w \in B(c,r)} \text{dist}(v, w).$$

Notice that $u' \in C_u$ and $v' \in C_v$ respectively. Since c is of distance at most r away from any vertex in $B(c, r)$,

$$\begin{aligned} \text{dist}_\delta(u, v') &= \text{dist}(u, v') - \delta_u \\ &\leq \text{dist}(u, c) + \text{dist}(c, v') - \delta_u \\ &\leq \text{dist}_\delta(u, c) + r. \end{aligned}$$

Similarly

$$\begin{aligned} \text{dist}_\delta(v, v') &= \text{dist}(v, v') - \delta_v \\ &\geq \text{dist}(v, c) - \text{dist}(c, v') - \delta_v \\ &\geq \text{dist}_\delta(v, c) - r. \end{aligned}$$

Since $v' \in C_v$, we must have $\text{dist}_\delta(v, v') \leq \text{dist}_\delta(u, v')$, therefore

$$\begin{aligned} \text{dist}_\delta(v, c) - r &\leq \text{dist}_\delta(u, c) + r \\ \text{dist}_\delta(v, c) - \text{dist}_\delta(v, c) &\leq 2r. \end{aligned}$$

The same argument can be applied to more than two intersecting clusters. When $B(c, r)$ intersects k or more clusters, the k smallest shifted distance from all the vertices to c must fall in a range of at most $2r$. We now give an upper bound on the probability of this happening.

For each vertex $v \in V$, let $X_v = -\text{dist}_\delta(v, c) = \delta_v - \text{dist}(v, c)$ be the negative shifted distance. Notice that each X_v is nothing more than an exponentially distributed random variable with some constant offset. Let S vary over subsets of V of size $k - 1$, let $u \in V \setminus S$ and $\alpha \in \mathbb{R}^+$. Denote by $E_{S,u,\alpha}$ the event that $Y_u = \alpha$, $Y_v \geq \alpha$ for all $v \in S$ and $Y_v < \alpha$ otherwise. In other words, the set S contains the $k - 1$ nearest vertices in terms of the shifted distance, with u being kth nearest at distance α away. The law of total probability gives that

$$\begin{aligned} &\mathbf{Pr}\left[B(c, r) \text{ intersects } k \text{ or more clusters}\right] \\ &= \sum_{\substack{S \subset V \\ |S| = k-1}} \sum_{u \in V \setminus S} \int_\alpha \mathbf{Pr}\left[Y_v \leq \alpha + 2r \text{ for all } v \in S \mid E_{S,u,\alpha}\right] \mathbf{Pr}\left[E_{S,u,\alpha}\right] \end{aligned}$$

Thus it suffices to show that

$$\mathbf{Pr}\left[Y_v \leq \alpha + 2r \text{ for all } v \in S \mid E_{S,u,\alpha}\right] \leq (1 - \exp(-2\beta r))^{k-1} \tag{2.1}$$

for any fixed S, u and α. For each $v \in S$,

$$\begin{aligned} & \mathbf{Pr}\left[Y_v \leq \alpha + 2r \mid \ Y_v \geq \alpha\right]. \\ = & \mathbf{Pr}\left[\delta_v \leq \alpha + 2r + \mathrm{dist}(v,c) \mid \delta_v \geq \alpha + \mathrm{dist}(v,c)\right]. \end{aligned}$$

There are two cases to consider. If $\alpha + \mathrm{dist}(v,c) \leq 0$, then since the exponential distribution has non-negative support,

$$\begin{aligned} & \mathbf{Pr}\left[\delta_v \leq \alpha + 2r + \mathrm{dist}(v,c) \mid \delta_v \geq \alpha + \mathrm{dist}(v,c)\right]. \\ \leq & \mathbf{Pr}\left[\delta_v \leq 2r\right] \\ = & 1 - \exp(-2\beta r). \end{aligned}$$

If $\alpha + \mathrm{dist}(v,c) > 0$, then by the memoryless property of the exponential distribution, we have

$$\begin{aligned} & \mathbf{Pr}\left[\delta_v \leq \alpha + 2r + \mathrm{dist}(v,c) \mid \delta_v \geq \alpha + \mathrm{dist}(v,c)\right]. \\ = & \mathbf{Pr}\left[\delta_v \leq 2r\right] \\ = & 1 - \exp(-2\beta r). \end{aligned}$$

Since the Y_v's are independent, we have

$$\begin{aligned} \mathbf{Pr}\left[Y_v \leq \alpha + 2r \text{ for all } v \in S \mid E_{S,u,\alpha}\right] &= \prod_{v \in S} \mathbf{Pr}\left[Y_v \leq \alpha + 2r \mid \ Y_v \geq \alpha\right] \\ &= (1 - \exp(-2\beta r))^{k-1}. \end{aligned}$$

■

This lemma is more general then the bound on edge-cutting probability that we aimed for in Definition 2.1.1, but is needed in our parallel construction of spanners in Chapter 3. Upper bound on the probability of cutting an edge can be obtained by applying this lemma with the midpoint of the edge as the center c and letting $k = 2$. Here the distances $\mathrm{dist}(c, \cdot)$ is defined in the natural way (i.e. as if the edge is subdivided into two new edges with half the original length).

Corollary 2.2.4. *Any edge e with length $l(e)$ is cut in the clustering produced by* EST-Clustering *with probability at most* $1 - \exp(-\beta \cdot l(e)) < \beta \cdot l(e)$.

Combining Lemma 2.2.2 and Corollary 2.2.4, we obtain the following theorem on the correctness of our algorithm.

Theorem 2.2.5. *For any graph G and parameter β,* EST-CLUSTERING *produces a* $(\beta, O(\frac{\log n}{\beta}))$*-decomposition of G with high probability.*

We can also use Corollary 2.2.4 to prove another theorem on the number of clusters produced on unweighted graphs.

Theorem 2.2.6. *Given any connected unweighted graph G on n vertices,* EST-CLUSTERING *produces at most $O((1 - \exp(-\beta))n)$ clusters in expectation.*

Proof. Let T be an arbitrary spanning tree of G. We consider the clustering produced by running EST-CLUSTERING on G, but focus our attention on the edges in T. In expectation, $(1 - \exp(-\beta))(n - 1)$ edges from T are cut by Corollary 2.2.4, thus breaking T into at most $O((1 - \exp(-\beta))n)$ connected components in expectation. Any such connected component must also be connected in G, as T is a subgraph. Thus we get that the clustering contains $O((1 - \exp(-\beta))n)$ clusters in G.

■

2.3 Parallel Implementation on Unweighted Graphs

In this section we discuss how to implement the EST-CLUSTERING algorithm in $O(\frac{\log n}{\beta})$ parallel depth and $O(m)$ work on unweighted graphs. The formulation of Algorithm 1, while mathematically clean, is not readily amenable for implementation. We give an alternative formulation in Algorithm 2 which should shed more light on its parallel and distributed nature.

Algorithm 2 EST-CLUSTERING-IMPL(G, β)

Input: Weighted graph $G = (V, E, w)$ and parameter β.
Output: A $(\beta, O(\frac{\log n}{\beta}))$-decomposition of G with high probability

1: sample independent random shifts $\delta_v \sim \text{Exp}(\beta)$ for each $v \in V$
2: compute $\delta_{\max} = \max_{v \in V} \delta_v$
3: add a super source s to G and edges from s to each vertex v with length $\lceil \delta_{\max} \rceil - \delta_v$
4: compute the shortest path tree T from s using parallel BFS, breaking ties lexicographically
5: **return** the subtrees rooted at the children of s

Our first observation is that the shift in start time at each vertex v can be simulated by adding a super source s with an edge to v with length equal to the shift. We then compute the shortest path tree rooted at s and return the subtrees of s as the clusters. We can further add $\lceil \delta_{max} \rceil$ to the edges incident to s to make the lengths non-negative.

However it is still unclear how to compute such a tree in only $O(m)$ work when the graph is sparse, as it seems this would effectively sort the n shift values. The crucial fact here is that

the shift values are independent and exponentially distributed. The following is a well known fact about the gap between consecutive order statistics for exponential random variables.

Fact 2.3.1. *Let $X_1, X_2, \ldots, X_n$ be n independent and identically distributed exponential random variable with rate parameter λ and let $X_{(1)} < X_{(2)} < \cdots < X_{(n)}$ be the order statistics. Let $Y_1 = X_{(1)}$ and $Y_i = X_{(i)} - X_{(i-1)}$ for $i = 2, 3, \ldots, n$. Then Y_i is a random variable of the exponential distribution with rate parameter $(n - i + 1)\lambda$.*

Proof. Let F_{Y_1} be the cumulative distribution function of the smallest order statistic $Y_1 = X_{(1)}$. By the independence of the X_is, we have

$$\begin{aligned} 1 - F_{Y_1}(y) &= \mathbf{Pr}\left[Y_1 \geq y\right] \\ &= \mathbf{Pr}\left[X_1 \geq y, X_2 \geq y, \ldots, X_n \geq y\right]. \end{aligned}$$

Since the X_is are i.i.d. exponential random variables with rate parameter λ, this probability is $(\exp(-\lambda y))^n = \exp(-n\lambda y)$. Therefore

$$F_{Y_1}(y) = 1 - \exp(-n\lambda y)$$

and Y_1 is an exponential random variable with rate parameter $n\lambda$. Then applying the memoryless property, we see that $Y_2 = X_{(2)} - X_{(1)}$ has the same distribution as the first order statistic of $n - 1$ fresh i.i.d. exponential random variables. Therefore our claim for $Y_2, Y_3, \ldots, Y_n$ can be proved by repeating this argument inductively.

■

We are now ready to complete our analysis of the exponential start time clustering algorithm.

Theorem 2.3.2. *Given an unweighted graph G with n vertices and m edges, and any parameter $\beta > 0$,* EST-Clustering-Impl *can be implemented in $O(\frac{\log n \log^* n}{\beta})$ depth with high probability and $O(m)$ work.*

Proof. We first use Fact 2.3.1 to generate the gaps between order statistics of n exponential random variables with rate parameter β. Then applying parallel algorithms for computing prefix sums ([Rei93], Chapter 1), we can generate the shift values in sorted order in $O(\log n)$ depth and $O(n)$ work. We can then generate a random permutation of the vertices in parallel in $O(\log n)$ depth and $O(n)$ work (see [MR85, Rei85]), and assign the sorted shift values to this random permutation.

In order to implement the single source shortest path search from s using parallel BFS, notice that only the edges leaving s have non-integral lengths. Therefore the fractional part of the distance from s to a vertex v only depends on the shift value of the next vertex on the

shortest path from s to v. Since the only time we need to examine this fractional part during the search is to break ties when the integer distances are equal, we can replace the fractional parts in the shift values by breaking ties according a random permutation. This is once again a consequence of exponential distribution's memoryless property (applied to the fractional parts).

At this point we reduced the problem to a graph with unit length edges except those leaving the source s, which have integral lengths. We handle this by maintaining two queues: one regular BFS queue and one queue with the sorted shift values. In each step of the BFS, the smaller distance out of the two queues are chosen, and an edge from s to v with length l is only processed the if BFS frontier has expanded to distance l and v has not been visited yet.

We now bound the cost of running parallel BFS on our augmented graph. Lemma 2.2.2 shows that the BFS only need to search out $O(\frac{\log n}{\beta})$ levels with high probability. Each level of the search can be parallelized in $O(\log^* n)$ overhead in the depth with the CRCW PRAM model while the total work is $O(m)$ [GMV91, KS97]. We remark that this factor of $O(\log^* n)$ depends on the particular model of parallelism, for example it is $O(1)$ in the OR CRCW PRAM model, but can be bounded by $O(\log n)$ in most models.

Since the cost of this algorithm is dominated by the BFS step, and the overall depth is $O(\frac{\log n \log^* n}{\beta})$ with high probability and the total work is $O(m)$.

■

2.3.1 Remarks

Since the clusters are generated from a BFS computation, we obtain for free a spanning tree with small diameter on each cluster. These trees are crucial in the applications to finding graph spanners and low stretch trees in subsequent chapters, thus we make this observation formal in the following.

Corollary 2.3.3. *In addition to the low diameter clustering,* EST-Clustering-Impl *also produces a spanning tree on each of the output cluster. The diameter of each such tree is also $O(\frac{\log n}{\beta})$ with high probability.*

Laxman *et al.* [SDB14] gave an efficient parallel implementation of the exponential start time clustering and applied it to parallel algorithms for graph connectivity. They also showed that the lexicographical tie breaking from EST-Clustering-Impl can be replaced by arbitrary tie breaking with only a constant effect on the probabilistic guarantee of the algorithm, further simplifying the implementation and improving the performance in practice.

The readers might have noticed that we have only presented an implementation of our clustering algorithm in unweighted graphs. This is because with general edge lengths, the

frontier of the shortest path search no longer advances one edge at a time, but rather by an increment that is related to the shortest edge length. Therefore in order to upper bound the depth of the search, we usually resort to techniques such as grouping edges by length and apply the unweighted algorithm on each group, or rounding edge lengths. The particular ways we achieve this in different applications will be discussed in detail in the subsequent chapters.

We also point out that the exponential start time clustering is also inherently a distributed algorithm. It has an obvious implementation in the CONGEST model, and we refer the readers to [EN16] for a recent application in the distributed setting.

Chapter 3

Parallel Graph Spanners and Combinatorial Sparsifiers

3.1 Introduction

In this section we use the exponential start time clustering algorithm from Chapter 2 to give work efficient parallel algorithms for constructing graph spanners and sparsifiers.

Spanners are sparse structures that approximately preserve the shortest path metric of the original graph. Formally, we define a (multiplicative) spanner to be a sparse subgraph that approximates all-pair distances of the original graph up to some stretch factor. Given a graph $G = (V, E, l)$ with positive edge lengths, a k-spanner $H = (V, E', l)$ where $E' \subseteq E$, is a subgraph of G such that for any $u, v \in V$, $\text{dist}_G(u, v) \leq k \cdot \text{dist}_H(u, v)$. Here $\text{dist}_G(\cdot, \cdot)$ and $\text{dist}_H(\cdot, \cdot)$ refer to the shortest path distances in G and H respectively.

Given a target stretch factor, the goal is then to find spanners with fewest possible number of edges. Peleg and Schäffer [PS89] introduced this graph theoretic concept and gave a linear time sequential algorithm for $(2k-1)$-spanners in unweighted graphs with $\frac{1}{2}n^{1+1/k}$ edges, for any parameter k. This result was later extended by Althöfer *et al.* [ADD+93] to weighted graphs. However, their algorithm requires the solution of an incremental dynamic shortest path problem, and is much more is more expensive than the nearly linear work algorithms we are going to present. This trade-off between the stretch factor and the spanner size is essentially optimal up to a conjecture on graph girth by Erdős [Erd64]. Namely, it is conjectured that there exists graphs with $\Omega(n^{1+1/k})$ edges and girth greater than $2k$, and consequently these graphs only admits themselves as $(2k-1)$-spanners.

Spanners have numerous applications in distributed computing [Awe85, PU89], approximating shortest path distances [ABCP98, Coh98, TZ05], and also graph sparsification [Kou14,

unweighted graphs

Stretch	Size	Work	Depth	Notes
$O(2^{\log^* n} \log n)$	$O(n)$	$O(m \log n)$	$O(\log n \log^* n)$	[Pet10], distributed
$O(k)$	$O(n^{1+1/k})$	$O(m)$	$O(k \log^* n)$	[MPVX15], distributed

weighted graphs

Stretch	Size	Work	Depth	Notes
$2k-1$	$O(kn^{1+1/k})$	$O(km)$	$O(k \log^* n)$	[BS07]
$O(k)$	$O(n^{1+1/k} \log k)$	$O(m)$	$O(k \log^* n \log U)$	[MPVX15]

Figure 3.1: Known results on parallel algorithms for spanners, where $U = \frac{\max_e w(e)}{\min_e w(e)}$.

KX16], which is discussed in more detail in Section 3.3.

Figure 3.1 gives a comparison between previous results and our algorithm. For any parameter k, our construction produces $O(k)$-spanners of size $O(n^{1+1/k})$ on unweighted grpahs. This improves upon previous works by a factor of $O(k)$ on the spanner size, while losing a constant factor in the stretch. For weighted graphs, the improvement on the size of the spanner becomes a factor of $O(k/\log k)$ and the depth depends on the ratio between the longest and the shortest edge lengths. As we will see in Section 3.3, when we apply our spanners to spectral sparsification, the constant factor loss in the stretch is outweighted by the improvement in the spanner size. The results listed for unweighted graphs both happen to be also distributed, with the number of rounds equal to the parallel depth.

A spectral sparsifier of graph G is a sparse subgraph graph H such that

$$(1-\epsilon)x^T \mathbf{L}_G x \leq x^T \mathbf{L}_H x \leq (1+\epsilon)x^T \mathbf{L}_G x, \qquad \forall x \in \mathbb{R}^n$$

for some error parameter ϵ. For convenience, we will also write this guarantee as

$$(1-\epsilon)G \preceq H \preceq (1+\epsilon)G.$$

It follows that a spectral sparsifier shares most of the spectral properties of the original graph despite having fewer edges, allowing it to substitute the original graph in many applications in order to save computation (see for example [SWT16]).

Spectral sparsification of graphs was introduced by Spielman and Teng as a component in the first nearly-linear time SDD linear system solver [ST13, ST11, ST14], and keeps to play a crucial role in the subsequent solvers [KMP14, KMP11, CKM+14]. Spielman and Teng's sparsification algorithm is combinatorial in nature, it relies on intricate graph partitioning followed by uniform sampling in some of the partitions. Unfortunately, this produces a sparsifier of size $O(n \log^c n/\epsilon^2)$ for a fairly large constant c. Spielman and Srivastava [SS11] later introduced an elegant construction of spectral sparsifiers with only $O(n \log n/\epsilon^2)$ edges based on sampling

edges with effective resistances. However, in order to estimate these effective resistances, their algorithm requires $O(\log n)$ linear system solves in the graph Laplacian.

Recent efforts have been made in trying to design *combinatorial* algorithms for graph sparsification [KP12]. In [Kou14] Koutis showed how to compute estimates of effective resistances by repeatedly finding spanners with $O(\log n)$ stretch, and gave a nearly-linear work parallel combinatorial algorithm for constructing sparsifiers of size $O(s \log^2 n \log^2 \rho / \epsilon^2 + m/\rho)$, where s is the size of the spanners used and ρ is a parameter.

In Section 3.3 we apply our spanner algorithm and tighten the "resistances in parallel" argument from [Kou14] to obtain the a parallel and combinatorial algorithm for spectrally sparsifying a graph down to $O(n \log^2 n / \epsilon^2)$ edges.

3.2 Spanners

3.2.1 The Unweighted Case

Our spanner construction for unweighted graphs is given in Algorithm 3. It has the same structure as the original sequential algorithm by Peleg and Schaffer [PS89]: after the low diameter decomposition step, an edge is added between each pair of adjacent clusters.

Algorithm 3 UNWEIGHTED-SPANNER(G, k)

Input: Unweighted graph $G = (V, E)$ and parameter $k \geq 1$

1: $H \leftarrow$ ESTCLUSTER$(G, \log n / 2k)$ (Recall ESTCLUSTER returns a spanning tree on each cluster)
2: **for** each pair of adjacent cluster **do**
3: add an (arbitrary) edge between them to H
4: **end for**
5: **return** H

The Peleg and Schaffer algorithm [PS89] relied on a bound by Awerbuch [Awe85], which bounds the number of clusters interacting with the neighborhood of a single vertex. The same bound can be obtained with our exponential start time clustering algorithm using Lemma 2.2.3.

Lemma 3.2.1. *Given an unweighted graph $G = (V, E)$ and parameter $\beta = \frac{\log n}{2k}$,* ESTCLUSTER *produces a clustering of G such that for any vertex $v \in V$, the neighborhood of v, $B(v, 1) = \{u \in V \mid d(u, v) \leq 1\}$, intersects $O(n^{1/k})$ clusters in expectation.*

Proof. By Lemma 2.2.3, $B(v, 1)$ intersects k or more clusters with probability at most $(1 -$

$\exp(2\beta))^{k-1}$. Let L be the number of clusters intersecting $B(v, 1)$, we then have

$$\begin{aligned}
\mathbb{E}[L] &= \sum_{l=1}^{\infty} \mathbf{Pr}[L \geq l] \\
&\leq \sum_{l=1}^{\infty} (1 - \exp(-2\beta))^{l-1} \\
&= \frac{1}{\exp(-2\beta)} \\
&= \frac{1}{\exp(-\log n/k)} \\
&= n^{1/k}.
\end{aligned}$$

■

Lemma 3.2.2. *Given a connected unweighted graph and for any $k \geq 1$,* Unweighted-Spanner *constructs a $O(k)$-spanner with high probability with expected size $O(n^{1+1/k})$. Furthermore this runs in $O(k \log^* n)$ depth with high probability and does $O(m)$ work.*

Proof. We first upper bound the size of the output graph. The algorithm starts by constructing an exponential start time clustering with parameter $\beta = \frac{\log n}{2k}$. Let H be the spanning forest obtained from the decomposition, notice that H has at most $n - 1$ edges. In the second phase where edges between adjacent clusters are added to H, the contribution to the total size can be bounded on a per vertex basis. Consider a boundary vertex v, (i.e. v is incident to an inter-cluster edge), we add at most one edge from v to each of the clusters adjacent to v. Applying Lemma 3.2.1 to the neighborhood of each $v \in V$, we see that in expectation at most $O(n^{1+1/k})$ edges are added this way.

It remains to bound the stretch of edges that are not included in the spanner. For an edge $e \notin H$ internal to a cluster, its stretch is certified by the spanning tree within the cluster. In particular the tree path between the endpoints of e is at most the diameter of the cluster, which bounded by $O(k)$ with high probability according to Lemma 2.2.2. For an edge $e \notin H$ whose endpoints are in two different clusters, our spanner must contain another edge e' between these two clusters. As with high probability both of these clusters has diameter $O(k)$, the stretch of e is again bounded by $O(k)$ with high probability.

The depth and work bounds follow from Theorem 2.3.2.

■

3.2.2 The Weighted Case

We first observe that our unweighted spanner construction can still produce good spanners on weighted graphs when the edge lengths are bounded within a constant range. Given $G = (V, E, l)$ where $U = (\max_e l(e))/(\min_e l(e))$ is the ratio between the maximum and the minimum edge lengths, we group the edges as

$$E_i = \{e \in E \mid l(e) \in [2^{i-1}, 2^i)\},$$

an run the unweighted algorithm on each E_i separately. Then taking the union of the results on each E_i will give as a spanner for G, but leads to an overhead of $O(\log U)$ in the spanner size.

We can reduce this overhead using a similar contraction scheme from Chapter 2 that allowed us to speed up Bartal's algorithm. We first reduce the problem to instances where edge can be grouped into buckets in which lengths differs significantly. We then build spanners on each of these buckets in order, but contract away the low-diameter components with smaller edge lengths. Lemma 3.2.2 then allows us to bound the expected rate at which vertices are contracted, and in turn the size of the spanner. This contraction scheme is significantly simpler the one in Chapter 2 ones because we only need to ensure that edge lengths in different buckets differ by factors of poly k, where k is the stretch factor.

Definition 3.2.3. *We say a weighted graph $G = (V, E, l)$ is δ-separated if we can partition the edge set as $E = E_1 \cup \cdots \cup E_t$ such that for any $i < j$, $e_i \in E_i$ and $e_j \in E_j$, we have $l(e_j) \geq \delta \cdot l(e_i)$.*

We first break up the input graph into $O(\log k)$ graphs where edge lengths are $O(k)$-separated. Let G_i to be the graph with vertex set V and edge set

$$\bigcup_{j \geq 0} E_{i+i \cdot c \lg k}.$$

By construction each G_i is a k^c-separated graph under Definition 3.2.3 and the union of $O(\log k)$ such G_is form the whole graph. The constant c here will be chosen to achieve the desired success probability from Theorem 2.2.5, we will hide c inside big-O notations from now on. Thus if we can find a $O(n^{1+1/k})$-sized spanner for each of the G_i, we will obtain a $O(n^{1+1/k} \log k)$-sized spanner for G. Pseudocode of this algorithm is given in Algorithm 4 and Algorithm 5.

Lemma 3.2.4. *Given a k^c-separated graph $G = (V, E, l)$,* Well-Separated-Spanner *constructs with high probability a $O(k)$-spanner for G with expected size $O(n^{1+1/k})$, in $O(k \log^* n \log U)$ depth and $O(m)$ work.*

Proof. For simplicity, we write $l(e) \in [l_i, l_{i+1})$ for any $e \in E_i$. In the ith iteration, the unweighted algorithm is run on the quotient graph $\Gamma_i = G[A_i]/H_{i-1}$. By Lemma 3.2.2 this produces an unweighted spanner for edges from E_i that are not contracted away in Γ_i. Since the edge weights differ by at least $O(k)$ between levels, using Lemma 2.2.2 and induction on the loop

Algorithm 4 WELL-SEPARATED-SPANNER(G)

Input: $O(k)$-separated graph $G = (V, E, l)$ with $E = E_1 \cup \cdots \cup E_s$ as in Definition 3.2.3

1: $H_0 \leftarrow \emptyset$ ▷ the H_is will guide the quotient graph contraction
2: $S \leftarrow \emptyset$
3: **for** $i = 1$ to s **do**
4: $\quad \Gamma_i \leftarrow G[A_i]/H_{i-1}$ with unit edge length
5: $\quad F_i \leftarrow \text{ESTCLUSTER}(\Gamma_i, \frac{\log n}{2k})$ ▷ recall F_i is a spanning forest of Γ_i's clusters
6: $\quad H_i \leftarrow H_{i-1} \cup F_i$
7: $\quad S \leftarrow S \cup F_i$
8: $\quad$ augment S by adding one edge between each pair of adjacent clusters in F_i
9: **end for**
10: **return** S

Algorithm 5 WEIGHTED-SPANNER(G)

Input: A weighted graph G

1: partition E in the $E_i = \{e \in E \mid l(e) \in [2^i, 2^{i+1})\}$
2: **for** $i = 1$ to $O(\log k)$, in parallel **do**
3: $\quad$ let $G_i = (V, \cup_{j \geq 0} E_{i+j \cdot c \lg k}, l)$
4: $\quad S_i \leftarrow \text{WELL-SEPARATED-SPANNER}(G_i)$
5: **end for**
6: **return** $\cup_i S_i$

index one can show that that vertices in the quotient graph Γ_i corresponds to pieces of diameter at most l_i in the spanner constructed so far, with high probability. Therefore the stretch bound for edges from E_i in Γ_i gets worse by at most a factor of 2 when translated in G.

It remains now to bound the size of the spanner, which amounts to bounding the total number of vertices with degree at least one in all of Γ_is, as singleton vertices do not contribute to the size bound from Lemma 3.2.2. Recall that each vertex in Γ_i corresponds to a cluster computed on Γ_{i-1}, it suffices to upper bound the overall number of clusters produced over the lifetime of the algorithm. By Theorem 2.2.6, the number of clusters produced on Γ_{i-1} (or equivalent the number of vertices in Γ_i) is at most $1 - \exp(-\beta)$ times the number of vertices in Γ_i in expectation. Since a singleton vertex in Γ_{i-1} can become connected by edges that were not previously considered in the ith round, it is easier to consider the contribution of each individual vertex and them sum over all the vertices. Initially there are n vertices in total, and the contribution from each of them starts out to be 1 and decrease geometrically. Therefore the total number of vertices in all of the Γ_is can be bounded by an argument similar to the proof of

Lemma 3.2.1:

$$\sum_{i=0}^{\infty}(1-\exp(-\beta))^i n = \left(\frac{1}{\exp(-\beta)}\right) n$$
$$= \left(\frac{1}{\exp(-\log n/2k)}\right) n$$
$$= n^{1+1/2k}.$$

By Lemma 3.2.1 each vertex contributes at most $O(n^{1/k})$ inter-cluster edges to the spanner, thus as stated WELL-SEPARATED-SPANNER produces a spanner of size $O(n^{1+3/2k})$, where the exponent $1+3/2k$ can be reduced to $1+1/k$ by slightly changing the value of β if we back down on the stretch bound by a factor of $3/2$.

For the depth and work bound, we will invoke Theorem 2.3.2. Notice that each iteration of the loop performs an exponential start time decomposition on disjoint sets of edges, therefore the overall work is $O(m)$. As there are $O(\log U)$ iterations, the overall depth is $O(k\log^* n\log U)$ with high probability.

■

Theorem 3.2.5. *Given a weighted graph G with n vertices, m edges and for any $k \geq 1$,* WEIGHTED-SPANNER *produces with high probability a $O(k)$-spanner for G of expected size $O(n^{1+1/k}\log k)$, in $O(k\log^* n\log U)$ depth and $O(m)$ work.*

Proof. WEIGHTED-SPANNER partitions the input graph into $O(\log k)$ edge disjoint subgraphs G_is which are individually processed by WELL-SEPARATED-SPANNER. By Lemma 3.2.4, this produce a $O(k)$-spanner of size $O(n^{1+1/k})$ on each of the G_i, and it is easy to see the union of them is also a $O(k)$-spanner of the original graph, with an overall size of $O(n^{1+1/k}\log k)$. Since WELL-SEPARATED-SPANNER processes each G_i in parallel, the overall depth and work also follow from Lemma 3.2.4.

■

3.3 Application to Combinatorial Sparsifiers

In this section we describe applications of our spanner algorithm to the problem of parallel and combinatorial graph sparsification. In [Kou14], Koutis gave a method for estimating effective resistances using graph spanners. Then combined with the method of oversampling by effective resistances [SS11, KMP14], this gives the following result on combinatorial graph sparsification.

Theorem 3.3.1 ([Kou14])**.** *Let G be a weighted graph with n vertices and m edges, and let* SPANNER *be an algorithm that can construct a $O(\log n)$-spanner of size S using D depth and W work in the CRCW*

PRAM model. Then for any parameters ϵ and ρ, there exists an algorithm Sparsify *that produces an output graph $\tilde{G}$ such that:*

1. *With high probability,*

$$(1-\epsilon)G \preceq \tilde{G} \preceq (1+\epsilon)G.$$

2. *The expected number of edges in $\tilde{G}$ is*

$$O\left(\frac{S\log^2 n \log^2 \rho}{\epsilon^2} + \frac{m}{\rho}\right).$$

3. Sparsify *can be implemented in the CRCW PRAM model with depth*

$$O\left(\frac{D\log^2 n \log^3 \rho}{\epsilon^2}\right)$$

   and work

$$O\left(\frac{W\log^2 n \log^2 \rho}{\epsilon^2}\right).$$

Our spanner algorithm from the previous section can be directly applied and improves the sparsifier size by a factor of $O(\log n)$ for unweighted graphs and $O(\log n/\log\log n)$ for weighted graphs (see Figure 3.1). We further improve on this by making an observation that tightens the analysis of [Kou14] for graph that are already sparse. First, we need to recall the following from [Kou14].

Definition 3.3.2. *Given a graph G and integer $t \geq 1$, a t-bundle k-spanner is a subgraph $H = H_1 \cup \cdots \cup H_t$ such that H_i is a k-spanner of $G \setminus \cup_{j=1}^{i-1} H_i$.*

Given a graph G, by Rayleigh's law of monotonicity, the effective resistance of an edge e can be bounded by its effective resistance in any subgraph H of G. Let H be a t-bundle k-spanner of G, by definition H contains t edge-disjoint paths of stretch at most k between the endpoints of e. Let H' be the subgraph consisting only of these edge disjoint paths, we can then upper bound the effective resistance between e's endpoints in H' (and therefore in H and G) by viewing them as resistors in parallel. This is summarized in the following lemma.

Lemma 3.3.3. *Let G be a graph and let H be a $2t$-bundle k-spanner of G. Then for every edge e in G that is not in H,*

$$w(e)\mathrm{ER}_G(e) \leq \frac{k}{t}.$$

We are now ready describe our improvement. At the core of Koutis [Kou14] is a sparsification routine that halves the size of the input graph. Given an error parameter ϵ, one first constructs and removes a $O(\frac{\log^2 n}{\epsilon^2})$-bundle $O(\log n)$-spanner from the input graph. Using Lemma 3.3.3, Koutis obtains good upper bounds on the effective resistances of the non-spanner edges, which allow these edges to be sampled into the sparsifier. On the other hand, the effective resistances of spanner edges have not been analyzed, and these edges are always kept in the sparsifier. We observe that these spanner edges in fact admit good upper bounds on their effective resistances, which gets better as the size of the spanner bundle increases. We formalize this observation in Algorithm 6

Algorithm 6 FURTHER-SPARSIFY$(G, \epsilon,$ SPANNER$)$

Input: Graph G, parameter $\epsilon > 0$, and a subroutine SPANNER for constructing $O(\log n)$-spanners.

1: $G_1 \leftarrow G$
2: $i \leftarrow 1$
3: $u_1 \leftarrow 1$
4: **while** G_i is non-empty **do**
5:     $H_i \leftarrow$ SPANNER(G_i)
6:     **if** $|H_i| < n$ **then**
7:         add arbitrary edges from G_i to H_i until $|H_i| = n$
8:     **end if**
9:     $G_{i+1} \leftarrow G_i \setminus H_i$
10:     $u_{i+1} \leftarrow \frac{\log n}{i}$
11:     $i \leftarrow i + 1$
12: **end while**
13: $\tilde{G} \leftarrow H_1$
14: over-sample G to produce $\tilde{G}$, using u_i as the sampling bias for edges in H_i
15: **return** $\tilde{G}$

Lemma 3.3.4. *Given a graph $G = (V, E, w)$ with n vertices and m edges, and an error parameter ϵ, assuming we have an algorithm* SPANNER *that produces $O(\log n)$-spanners of at most S edges,* FURTHER-SPARSIFY *produces a $(1 \pm \epsilon)$-spectral sparsifier $\tilde{G}$ such that:*

1. *With high probability,*

$$(1-\epsilon)G \preceq \tilde{G} \preceq (1+\epsilon)G.$$

2. *The expected number of edges in $\tilde{G}$ is*

$$O\left(\frac{S \log^2 n \log \frac{m}{n}}{\epsilon^2}\right).$$

Furthermore, if Spanner *has depth D and work W,* Further-Sparsify *runs in $O(\frac{m}{n}D)$ depth and $O(\frac{m}{n}W)$ work.*

Proof. As in the pseudocode, let H_i be the set of spanner edges returned by the spanner algorithm in the ith iteration of the loop. By Lemma 3.3.3, the $w(e)\mathrm{ER}(e)$ for any $e \in H_i$ is at most $u_i = \frac{k}{i-1}$ for $i \geq 2$, while the trivial bound $w(e)\mathrm{ER}(e) \leq 1$ is used for $e \in H_1$. Since the loop runs for at most m/n iterations and $n \leq |H_i| \leq S$, we have

$$\begin{aligned}\sum_{e \in E} w(e)\mathrm{ER}(e) &= S + S \sum_{i=2}^{m/n} \frac{O(\log n)}{i-1} \\ &= O(S \log n \log \frac{m}{n}).\end{aligned}$$

Then using the oversampling lemma from [KMP14], we can obtain a sparsifier graph $\tilde{G}$ using these effective resistance upper bounds with at most $O((S \log^2 n \log \frac{m}{n})/\epsilon^2)$ edges. The bound on work and depth follows from the fact that Further-Sparsify runs for at most $\frac{m}{n}$ iterations. ■

The Further-Sparsify routine repeatedly constructs and remove sparse spanners until the graph is exhausted. While this gives progressively better effective resistance upper bounds, it can also be expensive when the graph is originally dense. Therefore we combine it with the Sparsify routine from [Kou14] to obtain our final sparsification algorithm.

Algorithm 7 Full-Sparsify$(G, \varepsilon,$ Spanner$)$

Input: Graph G, error parameter ϵ, and $O(\log n)$-spanner algorithm Spanner
Output: A $(1 \pm \epsilon)$-sparsifier $\tilde{G}$ of G

1: choose the largest ϵ' such that $(\epsilon')^2 + 2\epsilon' = \epsilon$
2: $G' \leftarrow$ Sparsify$(G, \epsilon', \rho = n)$
3: $\tilde{G} \leftarrow$ Further-Sparsify(G', ϵ')
4: **return** $\tilde{G}$

Theorem 3.3.5. *Given a graph G, an error parameter $0 < \epsilon < 1$, assuming an algorithm* Spanner *that produces $O(\log n)$-spanners of size S,* Full-Sparsify *produces a graph $\tilde{G}$ that:*

1. *With high probability,*

$$(1-\epsilon)G \preceq \tilde{G} \preceq (1+\epsilon)G.$$

2. *The expected number of edges in $\tilde{G}$ is*

$$O\left(\frac{S \log^2 n \left(\log \log n + \log \frac{1}{\epsilon}\right)}{\epsilon^2}\right).$$

Furthermore, if Spanner *has depth D and work W,* Full-Sparsify *runs in $O(D\log^5/\epsilon^2)$ depth and $O(W\log^4/\epsilon^2)$ work.*

Proof. We start by analyzing the quality of the output graph $\tilde{G}$ as spectral sparsifier of G. First Full-Sparsify invokes Sparsify from Theorem 3.3.1 to obtain an intermediate graph G_1, with

$$(1-\epsilon')G \preceq G' \preceq (1+\epsilon')G.$$

We then invoke Further-Sparsify on G' to obtain $\tilde{G}$. By Lemma 3.3.4, we have

$$(1-\epsilon')G' \preceq \tilde{G} \preceq (1+\epsilon')G'.$$

Since we have chosen ϵ' to satisfy $(\epsilon')^2 + 2\epsilon' = \epsilon$, we also get via simple algebra that

$$(1-\epsilon)G \preceq \tilde{G} \preceq (1+\epsilon)G.$$

Next we upper bound the size of our sparsifier. First notice that since $0 < \epsilon' < \epsilon < 1$, we have $\frac{1}{\epsilon'} = O(\frac{1}{\epsilon})$, thus we can ignore the difference in error parameters in our bounds. Since we set $\rho = n$, applying Theorem 3.3.1 the expected number of edges in G' is

$$O\left(\frac{S\log^4 n}{\epsilon^2}\right).$$

By Lemma 3.3.4, the expected number of edges in $\tilde{G}$ can be bounded by

$$O\left(\frac{S\log^2 n\left(\log\log n + \log\frac{1}{\epsilon}\right)}{\epsilon^2}\right).$$

Since both the work and depth of Full-Sparsify is are dominated by the first step where we invoke Sparsify, the claimed bounds follows from Theorem 3.3.1.

■

Chapter 4

Low Stretch Tree Embeddings

4.1 Introduction

Over the last few years substantial progress has been made on a large class of graph theoretic optimization problems. We saw improvements in the asymptotic sequential running time and parallelizations for approximate undirected maximum flow and minimum cut [Mad10, CKM+11, LRS13, KLOS14, She13], bipartite matching [Mad13], minimum cost maximum flow [DS08], minimum energy flows [ST14, KMP11, KOSZ13, CFM+14], and graph partitioning [She09, OSV12]. One common aspect of all these new algorithms is that they all explicitly or implicitly use low-stretch spanning trees.

The fastest known algorithm for generating these trees, due to Abraham and Neiman, runs in $O(m \log n \log\log n)$ time [AN12]. Among the problems listed above, this running time is currently only the bottleneck for the minimum energy flow problem and its dual, solving symmetric diagonally dominant linear systems. However, there is optimism that all of the above problems can be solved in $o(m \log n)$ time, in which case finding these trees becomes a bottleneck as well.

The main question we will address in this chapter is the construction even better trees in only $\tilde{O}(m)$ time. This will remove the tree construction obstacle from $o(m \log n)$ time algorithms for solving SDD systems, as well as other graph optimization problems. We introduce two relaxation on the dependency of low stretch spanning trees that can simplify and speed up their construction. Firstly, we allow additional vertices in the tree, leading to a Steiner tree. This avoids the need for the complex graph decomposition scheme of [AN12]. Secondly, we discount the cost of high-stretch edges in ways that more accurately reflect how these trees are used. This allows the algorithm to be more "forgetful," and is crucial to our speedup.

Throughout this chapter we let $G = (V, E, l)$ be a graph with vertex set V, edge set E and

edge length function $l : E \rightarrow \mathbb{R}^+$. We use $T = (V_T, E_T, l_T)$ to denote the tree that we are trying to find. In previous works on low stretch spanning trees, T was required to be a subgraph of G in the weighted sense. In other words, $V_T = V$, $E_T \subseteq E$, and $l_T(e) = l(e)$ for all $e \in E_T$. We relax this condition by only requiring edge lengths in T to be not too short with respect to G through the notion of embeddability, which we formalize in Section 4.2.

For a tree $T = (V_T, E_T, l_T)$, the stretch of an edge $e = uv$ with respect to T is

$$\text{str}_T(e) \stackrel{\text{def}}{=} \frac{l_T(u,v)}{l(e)},$$

where $l_T(u,v)$ is the length of the unique path between u and v in T. Previous tree embedding algorithms aim to pick a T such that the total stretch of all edges e in G is small [AKPW95, AN12]. A popular alternate goal is to show that the expected stretch of any edge is small, and these two definitions are closely related [AKPW95, CCG+98] . Our other crucial definition is the discounting of high stretches by adopting the notion of ℓ_p-stretch:

$$\text{str}_T^p(e) \stackrel{\text{def}}{=} (\text{str}_T(e))^p .$$

These two definitional changes greatly simplify the construction of low stretch embeddings. It also allows the combination of existing algorithms in a robust manner. Our algorithm is based on the bottom-up clustering algorithm used to generate AKPW low-stretch spanning trees [AKPW95], combined with the top-down decompositions common in recent algorithms [Bar96, EEST08, ABN08, AN12]. Its guarantees can be stated as follows:

Theorem 4.1.1. *Let $G = (V, E, d)$ be a weighted graph with n vertices and m edges. For any parameter p strictly between 0 and 1, we can construct a distribution over trees embeddable in G such that for any edge e its expected ℓ_p-stretch in a tree picked from this distribution is $O((\frac{1}{1-p})^2 \log^p n)$. Furthermore, a tree from this distribution can be picked in expected $O(\frac{1}{1-p} m \log\log n)$ time in the RAM model.*

We will formally define embeddability, as well as other notations, in Section 4.2. An overview of our algorithm for generating low ℓ_p-stretch embeddable trees is in Section 4.3. We expand on it using existing low-stretch embedding algorithms in mostly black-box manners in Section 4.4. Then in Section 4.5 we show a two-stage algorithm that combines bottom-up and top-down routines that gives our main result.

Although our algorithm runs in $O(m \log\log n)$ time, the running time is in the RAM model, and our algorithm calls a sorting subroutine. As sorting is used to approximately bucket the edge weights, this dependency is rather mild. If all edge lengths are between 1 and Δ, this process can be done in $O(m \log(\log \Delta))$ time in the pointer machine model, which is $O(m \log\log m)$ when $\Delta \leq m^{\text{poly}(\log m)}$. We suspect that there are pointer machine algorithms without even this mild dependence on Δ, and perhaps even algorithms that improve on the runtime of $O(m \log\log n)$. Less speculatively, we also believe that our two-stage approach of combining

bottom-up and top-down schemes can be applied with the decomposition scheme of [AN12] to generate actual spanning trees (as opposed to merely embeddable Steiner trees) with low ℓ_p-stretch. However, we do not have a rigorous analysis of this approach, which would presumably require a careful interplay with the radius-bounding arguments in that paper.

4.1.1 Related Works

Alon et al. [AKPW95] first proposed the notion of low stretch embeddings and gave a routine for constructing such trees. They showed that for any graph, there is a distribution over spanning trees such that the expected stretch of an edge is $\exp(O(\sqrt{\log n \log\log n}))$. Subsequently, results with improved expected stretch were obtained by returning an arbitrary tree metric instead of a spanning tree. The only requirement on requirement on these tree metrics is that they don't shorten distances from the original graph, and they may also include extra vertices. However, in contrast to the objects constructed in this paper, they do not necessarily fulfill the embeddability property. Bartal gave trees with expected stretch of $O(\log^2 n)$ [Bar96], and $O(\log n \log\log n)$ [Bar98]. Optimal trees with $O(\log n)$ stretches are given by Fakcharoenphol et al. [FRT04], and are known as the FRT trees. This guarantee can be written formally as

$$\mathbb{E}[\mathrm{str}_T(e)] \leq O(\log n).$$

Recent applications to SDD linear system solvers has led to renewed interest in finding spanning trees with improved stretch over AKPW trees. The first low stretch spanning tree with $\mathrm{poly}(\log n)$ stretch was given by Elkin et al. [EEST08]. Their algorithm returns a tree such that the expected stretch of any edge is $O(\log^2 n \log\log n)$, which has subsequently been improved to $O(\log n \log\log n (\log\log\log n)^3)$ by Abraham et al. [ABN08] and to $O(\log n \log\log n)$ by Abraham and Neiman [AN12].

Syntactically our guarantee is almost identical to the expected stretch bound above when p is a constant strictly less than 1:

$$\mathbb{E}[\mathrm{str}_T^p(e)] \leq O(\log^p n).$$

The power mean inequality implies that our embedding is weaker than those with ℓ_1-stretch bounds. However, at present, $O(\log n)$ guarantees for ℓ_1-stretch are *not known*–the closest being the result by Abraham and Neiman [AN12], which is off by a factor of $\log\log n$.

Structurally, the AKPW low-stretch spanning trees are constructed in a bottom-up manner based on repeated clusterings [AKPW95]. Subsequent methods are based on top down decompositions starting with the entire graph [Bar96]. Although clusterings are used implicitly in these algorithms, our result is the first that combines these bottom-up and top-down schemes.

4.1.2 Applications

The ℓ_p-stretch embeddable trees constructed in this paper can be used in all existing frameworks that reduce the size of graphs using low-stretch spanning trees. In Section 4.6, we show that the larger graph with augmented with our low stretch Steiner tree can lead to linear operators close to the graph Laplacian of the original graph. It allows us to use these trees in algorithms for solving linear systems in graph Laplacians, and in turn SDD linear systems. This analysis also generalizes to other convex norms, which means that our trees can be used in approximate flow [LS13, She13] and minimum cut [Mad10] algorithms.

Combining our algorithm with the recursive preconditioning framework by Koutis et al. [KMP11] leads to an algorithm that runs solves such a system to constant accuracy in $O(m \log n)$ time. These tree embeddings are also crucial in the recent faster solver by Cohen et al. [CKM+14], which runs in about $m \log^{1/2} n$ time. Parallelizations of it can be used can also lead to work-efficient parallel algorithms for solving SDD linear systems with depth of about $m^{1/3}$ [BGK+14], and in turn for spectral sparsification [SS11, KLP15]. For these parallel applications, ignoring a suitable fraction of the edges leads to a simpler algorithm with lower depth. This variant of the algorithm is discussed in Section 4.5.3. On the other hand, these applications can be further improved by incorporating the recent polylog depth, nearly-linear work parallel solver by Peng and Spielman [PS14].

4.2 Embeddability

Before we describe our algorithm, we need to formally define the embeddability property that our trees satisfy. The definition we use is the same as the congestion/dilation definition widely used in routing [Lei92, LMR94]. It also appeared explicitly in earlier works on combinatorial preconditioning [Vai91, Gre96] and is implicit in the more recent algorithms.

Informally, an embedding of H into G generalizes the notion of H being a subgraph of G in the weighted sense. The vertex set of H is no longer required to be a subset of the vertex set of G, as long as it can be mapped into the latter (i.e. a vertex embedding). Then each edge $e \in H$ is embedded into G as a path connecting the images of e_H's endpoints under the vertex embedding.

Definition 4.2.1. *Given graphs $H = (V_H, E_H, l_H)$ and a graph $G = (V_G, E_G, l_G)$, a path embedding of H into G is characterized by the following three functions:*

1. *A function $\pi : V_H \to V_G$ that maps vertices of H to those in G.*
2. *A function $P : E_H \to \mathcal{P}(E_G)$, where $\mathcal{P}(S)$ denotes the power set of a set S. This function maps each edge $e_H \in E_H$ to a path of G. If u and v are the endpoints of e_H, then $P(e_H)$ is a path in G*

that goes from $\pi(u)$ to $\pi(v)$.

3. *A function $W : E_H \times E_G \to \mathbb{R}^+$, where $W(e_H, e_G) > 0$ if and only if $e_G \in P(e_H)$, in which case $W(e_H, e_G)$ represents the weight or the capacity of the edge e_G used to support the path embedding $P(e_H)$.*

Since an edge in G can be used in the path embedding of multiple edges in H, we can think of it as being allocated among the different paths. Intuitively, each allocation is less than the original edge, thus is harder to traverse in the graph connectivity sense, therefore its length should become longer. As a result, it is convenient to attribute an edge e both a length $l(e)$ and weight $w(e)$, which is the reciprocal of its length:

$$w(e) \stackrel{\text{def}}{=} \frac{1}{l(e)}.$$

We can think a path embedding of H into G as a way to support the connectivity of H using vertices and edges in G. If two vertices in H were connected by an edge e_H, then the length of the path $P(e_H)$ should not be too long compared to the original edge length $l_H(e_H)$. On the other hand, this is limited by the connectivity of G itself, since an edge e_G in G has its predetermined weight or capacity $w_G(e_G)$, to be split between all the path embeddings that uses e_G. To formalize the notion of H can be well supported by G, we use congestion-dilation definition of embeddability:

Definition 4.2.2. *A graph H is* path embeddable*, or simply* embeddable*, into a graph G, if there exists a path embedding (π, P, W) of H into G such that:*

- *For all edges $e_G \in E_G$,*

$$\sum_{e_H \in E_H} W(e_H, e_G) \leq w_G(e_G).$$

  In other words, the congestion on e_G is at most one.

- *For all edges $e_H \in E_H$,*

$$\sum_{e_G \in P(e_H)} \frac{1}{W(e_H, e_G)} \leq l_H(e) = \frac{1}{w_H(e)}.$$

  In other words, the dilation of e_H is at most one.

The first condition states that we do note over-allocate the weight or capacity of any edge in G. The second condition states that each edge from H doesn't become longer when embedded as a path in G (recall that length of an edge is the inverse of its weight). Note that since G has no self-loops, the definition precludes mapping both endpoints of an edge in H to the same point in G. Also if H is a subgraph of G and $w_H(e) \leq w_G(e)$ for all $e \in E_H \subseteq E_G$, setting π to be the identity function, $P(e) = \{e\}$, and $W(e, e) = w_H(e)$ is one way to certify embeddability.

4.3 Overview of the Algorithm

We now give a high level overview of our main results. Our algorithm follows the decomposition scheme taken by Bartal for generating low stretch embeddings [Bar96]. This scheme partitions the graph repeatedly to form a laminar decomposition, and then constructs a tree from the laminar decomposition. However, our algorithm also makes use of the spanning forests that were produced as a side product of the decomposition. Thus we start with the following alternate definition of Bartal decompositions where these trees are made explicit.

Definition 4.3.1. *Let $G = (V, E, l)$ be a connected multigraph. We say that a sequence of forests* $\mathbf{B}$*, where*

$$\mathbf{B} = ((B_0, l_{B_0}), (B_1, l_{B_1}), \ldots, (B_t, l_{B_t})),$$

is a Bartal decomposition *of G if all of the following conditions are satisfied:*

1. *Each $B_i \subseteq E$ is a forest graph equipped with the length function l_{B_i}. Furthermore B_0 is a spanning tree of G and B_t is the empty graph.*
2. *For any $i \leq t$, B_i is a subgraph of G in the weighted sense.*
3. *For any pair of vertices u, v and any $i < t$, if u and v are in the same connected component of B_{i+1}, then they are in the same connected component of B_i.*

If the length functions l_{B_i}s are the same as the length function of G, we will often omit them and simply write $\mathbf{B} = (B_0, \ldots, B_t)$. Condition 2 implies that each of the B_is is embeddable into G. A strengthening of this condition would require the union of all the B_is to be embeddable into G. We will call such decompositions *embeddable Bartal decompositions*.

Bartal decompositions correspond to laminar decompositions of the graphs: if any two vertices u and v are separated by the decomposition in level i, then they are also separated in all levels $j > i$. Given a laminar decomposition, Bartal's original algorithm [Bar96] then turns each connected components in the decomposition into a vertex, and connect a vertex u in level i with a vertex v in level $i + 1$ with an edge of length d_i if the corresponding component of v is contained in the component of u. This produces a Steiner tree whose leaves are the original vertices.

Given a sequence of non-negative real numbers $\mathbf{d} = (d_0, \ldots, d_t)$, we say that the sequence is geometrically decreasing if there exists some constant $0 < c < 1$ such that $d_{i+1} \leq c d_i$. Below we formalize the condition when such sequences can be used as diameter bounds for a Bartal decomposition.

Definition 4.3.2. *A geometrically decreasing sequence $\mathbf{d} = (d_0, \ldots, d_t)$ bounds the diameter of a Bartal decomposition $\mathcal{B}$ if for all $0 \leq i \leq t$,*

1. *The diameter of any connected component of B_i is at most d_i (with respect to the length function l_{B_i}).*
2. *Any edge $e \in B_i$ has length $l_{B_i}(e) \leq \frac{d_i}{\log n}$.*

If u and v are in the same partition in some level i, but are separated in level $i+1$, we say that u and v are first cut at level i. Given such a geometrically decreasing diameter bounds, the bound d_i for the level where an edge is first cut will dominate its final stretch under Bartal's tree construction. This motivates us to define the ℓ_p-stretch of an edge w.r.t. a Bartal decomposition as follows:

Definition 4.3.3. *Let $\mathbf{B}$ be a Bartal decomposition with diameter bounds $\mathbf{d}$, and let $p > 0$ be a parameter. Then the ℓ_p-*stretch *with respect to $\mathbf{B}$ and $\mathbf{d}$ of an edge e with length $l(e)$ that is first cut at level i is given by*

$$str^p_{\mathbf{B},\mathbf{d}}(e) \stackrel{def}{=} \left(\frac{d_i}{l(e)}\right)^p.$$

In Section 4.4, we will show that it suffices to generate a (not necessarily embeddable) Bartal decomposition for which edges are expected to have small ℓ_p-stretch, then apply a transformation to obtain an embeddable low stretch tree. We will give more details on this transformation later in this section as well.

The decomposition itself will be generated by repeatedly applying a variant of the probabilistic low-diameter decomposition from [Bar96], which we discussed in detail in Chapter 2. We rephrase the result in the following lemma.

Lemma 4.3.4 (Probabilistic Low Diameter Decomposition)**.** *There is an algorithm* Partition *that given a graph $G = (V, E, l)$ with n vertices and m edges, and a diameter parameter d, returns a partition of V into $V_1 \cup V_2 \cup \cdots \cup V_k$ such that:*

1. *The diameter of the subgraph induced on each V_i is at most d with high probability, certified by a shortest path tree on V_i with diameter d.*
2. *For any edge $e = \{u, v\}$ with length $l(e)$, the probability that u and v belong to different pieces is at most $O(\frac{l(e)\log n}{d})$.*

Furthermore, Partition *can be implemented by computing a single source shortest path tree on the same graph, from a super-source with edges to all other of length between 0 and d.*

At a high level, our algorithm first fixes a geometrically decreasing sequence $\mathbf{d}$, then recursively decomposes the graph using Partition from Lemma 4.3.4 using $\mathbf{d}$ as diameter bounds. With regular (ℓ_1) stretch, this scheme can be shown to give expected stretch of about $O(\log^2 n)$ per edge [Bar96], and most of the follow-up works focused on reducing this stretch factor. With ℓ_p-stretch on the other hand, such a trade-off is already sufficient for the optimum bounds when p is a constant bounded away from 1.

Lemma 4.3.5. *Let $\mathcal{B}$ be a distribution over Bartal decompositions. If $\mathbf{d}$ is a geometrically decreasing sequence that bounds the diameter of any $\mathbf{B} \sim \mathcal{B}$, and the probability of an edge with length $l(e)$ being cut on level i of some $\mathbf{B} \sim \mathcal{B}$ is*

$$O\left(\left(\frac{l(e)\log n}{d_i}\right)^q\right)$$

for some $0 < q \leq 1$. Then for any p such that $0 < p < q$, we have

$$\mathbb{E}\left[str^p_{\mathbf{B},\mathbf{d}}(e)\right] \leq O\left(\frac{1}{q-p}\log^p n\right).$$

The proof of this lemma relies on the following fact about geometric series, which plays a crucial role in all of our analyses.

Fact 4.3.6. *There is an absolute constant c_{geo} such that when given any $c \in [e, e^2]$ and $\epsilon > 0$, we have*

$$\sum_{i=0}^{\infty} c^{-i\epsilon} \leq \frac{c_{geo}}{\epsilon}.$$

Proof. Since $0 < c^{-1} < 1$, the sum converges, and equals to

$$\frac{1}{1-c^{-\epsilon}} = \frac{1}{1-\exp(-\epsilon \ln c)}.$$

Therefore it remains to lower bound the denominator. If $\epsilon \geq 1/4$, then the denominator can be bounded by a constant. Otherwise, we have $\epsilon \ln c \leq 1/2$, and using the fact that $\exp(-t) \leq 1 - t/2$ when $t \leq 1/2$ we obtain

$$1 - \exp(-\epsilon \ln c) \geq \epsilon \ln c.$$

Substituting in the bound on c and this lower bound into the denominator then gives the result.

■

Proof of Lemma 4.3.5. If an edge e is cut at a level with $d_i \leq l(e)\log n$, its stretch is at most $\log n$, giving an ℓ_p-stretch of at most $\log^p n$. Thus it suffices only to consider the event E_i where e is first cut at levels where $d_i \geq l(e)\log n$. Substituting the stretch bounds of an edge cut on level i and the probability of it being cut into the definition of ℓ_p-stretch gives:

$$\begin{aligned}
\mathbb{E}\left[\mathrm{str}^p_{\mathbf{B},\mathbf{d}}(e) \mid E_i\right] &\leq \sum_{\{i \mid d_i \geq l(e)\log n\}} \left(\frac{d_i}{l(e)}\right)^p O\left(\left(\frac{l(e)\log n}{d_i}\right)^q\right) \\
&= O\left(\log^p n \sum_{\{i \mid d_i \geq l(e)\log n\}} \left(\frac{l(e)\log n}{d_i}\right)^{q-p}\right).
\end{aligned}$$

Since $d_i \geq l(e) \log n$ and the d_is are geometrically increasing, this can be bounded by

$$\mathbb{E}\left[\mathrm{str}^p_{\mathbf{B},\mathbf{d}}(e) \mid E_i\right] \leq O\left(\log^p n \sum_{i=0} c^{-i(q-p)}\right)$$

Using Fact 4.3.6 and the law of total probability we obtain

$$\mathbb{E}\left[\mathrm{str}^p_{\mathbf{B},\mathbf{d}}(e)\right] \leq O\left(\frac{1}{q-p} \log^p n\right).$$

∎

This is our approach for showing that a Bartal decomposition has small ℓ_p-stretch, and it remains to convert them into embeddable trees. This conversion is done in two steps: we first show how to obtain a decomposition such that all of the B_is are embeddable into G, and then we give an algorithm for converting such a decomposition into an embeddable Steiner tree. To accomplish the former, we ensure that each B_i is embeddable by choosing them to be subgraphs. Then we present pre-processing and post-processing procedures that converts such a guarantee into embeddability of all the B_is simultaneously.

In order to obtain a tree from the decomposition, we treat each component in the laminar decomposition as a Steiner vertex, and join them using parts of the B_is. This step is similar to Bartal's construction in that it identifies centers for each of the B_is, and connects these centers between one level and the next. However, the need for our final tree to be embeddable means that we cannot use the star-topology from [Bar96]. Instead, we must use part of the B_is between the centers. As each B_i is a forest with up to n edges, a tree obtained as such may have a much larger number of Steiner vertices. As a result, the final step involves reducing the size of this tree by pruning and contracting the tree paths. This process is illustrated in Figure 4.1.

In Section 4.4, we give the details on these steps that converts Bartal decompositions to embeddable trees. We will show that Bartal's algorithm for generating these decompositions meets the cutting probability requirements of Lemma 4.3.5. This then gives the following intermediate result:

Lemma 4.3.7. *Given a graph G with weights are between $[1, \Delta]$, for the diameter sequence $\mathbf{d} = (d_0, d_1, \ldots, d_t)$ where $d_i = 2^{-(i+1)} n\Delta$ and $d_t < 1$, there is a distribution over Bartal decompositions with diameters bounded by $\mathbf{d}$ such that for any edge e and any parameter $0 < p < 1$,*

$$\mathbb{E}\left[str^p_{\mathbf{B},\mathbf{d}}(e)\right] \leq O\left(\frac{1}{1-p} \log^p n\right).$$

Furthermore, a random decomposition from this distribution can be sampled using Decompose-Simple *(Algorithm 8) with high probability in $O(m \log(n\Delta) \log n)$ time in the RAM model.*

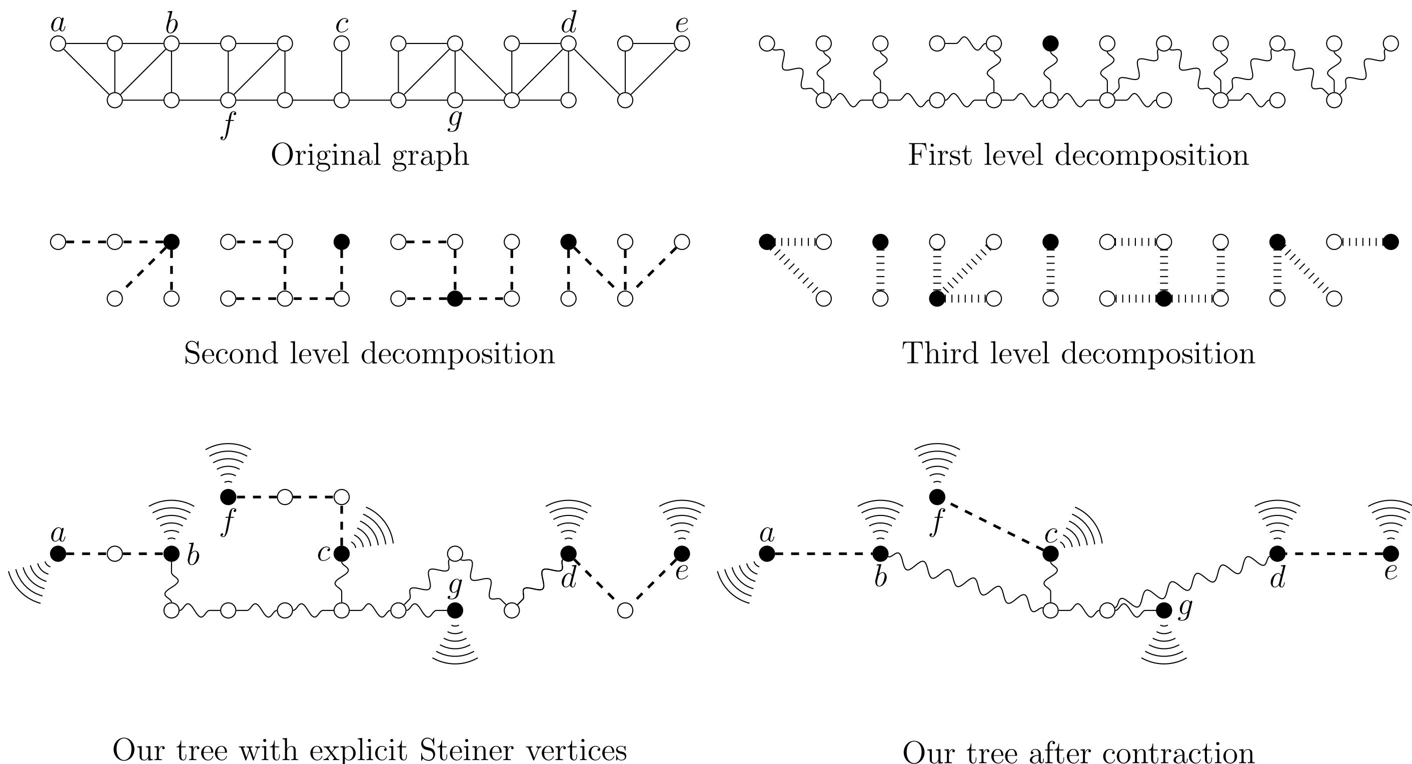


Figure 4.1: Bartal decomposition and the tree produced for a particular graph

This lemma combined with the embeddability transformations gives a simple algorithm for constructing low ℓ_p-stretch embeddable trees with expected stretch matching the bound stated in Theorem 4.1.1. However, the running time of $O(m \log(n\Delta) \log n)$ is more than the current best for finding low-stretch spanning trees [AN12], as well as the $O(m \log^2 n)$ running time for finding Bartal trees.

Our starting point towards a faster algorithm is the difference between our simplified routine and Bartal's algorithm. Bartal's algorithm, as well as subsequent algorithms [EEST08] ensure that an edge participates in only $O(\log n)$ decompositions. At each step, they work on a graph obtained by contracting all edges whose lengths are less than $d_i / \operatorname{poly}(n)$. This coupled with the upper bound of edge lengths from Definition 4.3.2 and the geometric decrease in diameter bounds guarantees that each edge is involved in $O(\log(\operatorname{poly}(n))) = O(\log n)$ levels of the decomposition.

As a path in the tree has at most n edges, the additive increase in stretch caused by these shrunken edges is negligible. Furthermore, the fact that our diameter decreases means that once we uncontract an edge, it remains uncontracted in all future steps. Therefore, these algorithms can start from the initial contraction for d_0, and maintain all contractions in work proportional to their total sizes.

When viewed by itself, this contraction scheme is almost identical to Kruskal's algorithm for building minimum spanning trees. This suggests that the contraction sequence can be viewed as another tree underlying the top-down decomposition algorithm. This leads to the question of whether other types of trees can be used in place of the minimum spanning tree. In Section 4.5, we show that if the AKPW low-stretch spanning tree can be used instead, and each edge is expected to participate in $O(\log\log n)$ levels of the top-down decomposition. Combining this with the $O(m\log\log n)$ time algorithm in the RAM model for finding the AKPW low-stretch spanning tree and a faster decomposition routine then leads to our faster algorithm.

Using these spanning trees to contract away parts of the graph leads to additional difficulties in the post-processing steps where we return embeddable Steiner trees. A single vertex in the contracted graph may correspond to a large cluster in the original graph. As a result, edges incident to it in the decomposition may need to be connected by long paths. Furthermore, the total size of these paths may be large, which means that they need to be treated implicitly. In Section 4.5.5, we leverage the tree structure of the contraction to implicitly compute the reduced tree. Combining it with the faster algorithm for generating Bartal decompositions leads to our final result as stated in Theorem 4.1.1.

4.4 Embeddable Trees from Bartal Decompositions

In this section, we show that embeddable trees can be obtained from Bartal decompositions using the process illustrated in Figure 4.1. We will achieve this in three steps: first presenting Bartal's algorithm using our language in Section 4.4.1, then showing that a decomposition routine that makes each B_i embeddable leads to a routine that generates embeddable decompositions in Section 4.4.2, and finally giving an algorithm for constructing a tree from the decomposition in Section 4.4.3.

4.4.1 Bartal's Algorithm

Bartal's algorithm in its simplest form can be viewed as repeatedly decomposing the graph so the pieces have the diameter guarantees specified by $\mathbf{d}$. At each step, it performs a low-diameter probabilistic decomposition from Lemma 4.3.4.

This routine was first introduced by Bartal to construct these decompositions. It and the low diameter decompositions that it's based on constructed each V_i in an iterative fashion. Miller et al. [MPX13] showed that a similar procedure can be viewed globally, leading to the implementation-independent view described above. A single invocation of Dijkstra's algorithm then allows one to obtain a running time of $O((m+n)\log n)$. This can be further sped up to $O(m+n\log n)$ using Fibonacci heaps due to Fredman and Tarjan [FT87], and to $O(m)$ in the

RAM model using a result by Thorup [Thoo0]. In our setting where approximate answers suffice, a running time of $O(m + n \log\log \Delta)$ was also obtained by Koutis et al. [KMP11]. As our faster algorithm only relies on the shortest paths algorithm in a more restricted setting, we can use the most basic $O(m \log n)$ bound for simplicity.

We can then obtain Bartal decompositions by invoking this routine recursively. Pseudocode of the algorithm is given in Algorithm 8. The output of this algorithm for a suitable diameter sequence gives us the decomposition stated in Lemma 4.3.7.

Algorithm 8 DECOMPOSE-SIMPLE(G)

Input: Weighted graph $G = (V, E, l)$, $1 \leq l(e) \leq \Delta$ for all $e \in E$

1: $B_0 \leftarrow$ shortest path tree from an arbitrary vertex
2: generate a sequence $\mathbf{d} = (d_0, \ldots, d_t)$ such that $d_i = 2^{-(i+1)} n\Delta$ and $d_t < 1$.
3: **for** $i = 1, \ldots, t$ **do**
4: $\quad B_i \leftarrow \varnothing$
5: $\quad$ remove all edges e with $l(e) \geq \frac{d_i}{\log n}$ from G
6: $\quad$ **for** each subgraph H of G induced by a connected component of B_{i-1} **do**
7: $\quad\quad G_1, \ldots, G_k \leftarrow$ PARTITION(H, d_i)
8: $\quad\quad$ add the shortest path tree on each G_j to B_i
9: $\quad$ **end for**
10: **end for**
11: $\mathbf{B} \leftarrow ((B_0, l), (B_1, l), \ldots, (B_t, l))$, i.e. the lengths in each B_i are the same as in G
12: **return** $(\mathbf{B}, \mathbf{d})$

Proof of Lemma 4.3.7. Let $\mathbf{B}$ be the output of DECOMPOSE-SIMPLE$(G, \mathbf{d})$. Each tree in B_i is a shortest path tree on a cluster of vertices returned by the PARTITION routine. Since these clusters are disjoint, each B_i is a subgraph of G in the weighted sense. As the algorithm only refines the partitions, once two vertices are separated, they will remain separated in any further levels. Also B_0 is a spanning tree by construction, and B_t cannot contain any edge since any we assume edge lengths are at least 1 and $d_t < 1$. Thus $\mathbf{B}$ is a Bartal decomposition of G.

We now show that the sequence $\mathbf{d}$ gives valid diameter bounds for any decomposition produced with high probability. The diameter bounds on B_i for $i > 0$ follow from Lemma 4.3.4. Since B_0 is a shortest path tree on G, its diameter is also at most $d_0 = 2n\Delta$. By construction, for any $i > 0$ and $e \in B_i$, we have $l(e) \leq \frac{d_i}{\log n}$ as edges violating this constraint are discarded before each call to PARTITION. This constraint is also satisfied for $i = 0$ since $d_0 = 2n\Delta$ and $l(e) \leq \Delta$.

The cost of each iteration of the loop is dominated by the shortest path computation that implements PARTITION, since $t \leq O(\log(n\Delta))$ the running time of is algorithm can be bounded by $O(m \log n \log(n\Delta))$.

It remains to bound the expected ℓ_p-stretch of an edge e with respect to the decomposition.

When $l(e) \geq \frac{d_i}{c \log n}$, a suitable choice of constants allows us to bound the probability of e being cut by 1. Otherwise, e will not be removed unless it is already cut. In case that it is present in the graph passed onto Partition, the probability then follows from Lemma 4.3.4. Hence the cutting probability of edges satisfies Lemma 4.3.5, which gives us the bound on stretch.

■

4.4.2 Embeddability by Switching Moments

We now describe how to construct *embeddable* Bartal decomposition by using the routine from the previous section. This is done in three steps: first we pre-process the input graph G and scale the edge lengths of G to form G', then we run the decomposition routine on G' using a different parameter q, and finally we apply a post-processing step to convert the output into an embeddable Bartal decomposition. Pseudocode of this conversation procedure is given in Algorithm 9. Both the pre-processing and post-processing steps are deterministic, linear mappings. As a result, we can focus on bounding the expected stretch of an edge in the decomposition given by Decompose.

Algorithm 9 Embeddable-Decompose(G, p, q)

Input: Weighted graph $G = (V, E, l)$, $0 < p < q < 1$, and a decomposition routine Decompose_q.

1: $G' \leftarrow (V, E, l')$ where $l'(e) = l(e)^{\frac{q}{p}}$

2: $\mathbf{B}', \mathbf{d}' \leftarrow \text{Decompose}_q(G')$

3: create $\mathbf{B}$ and $\mathbf{d}$ from $\mathbf{B}'$ and $\mathbf{d}'$ by scaling the edge lengths and diameter bounds in each level i by

$$\frac{c_{geo}}{q-p}\left(\frac{d'_i}{\log n}\right)^{\frac{q-p}{p}}$$

where c_{geo} is the constant from Fact 4.3.6.

4: **return B, d**

We first verify that $\mathbf{d}$ is a geometrically decreasing sequence bounding the diameters of $\mathbf{B}$.

Lemma 4.4.1. *If $\mathbf{B}'$ is a Bartal decomposition of G' whose diameters are bounded by $\mathbf{d}'$, then $\mathbf{d}$ is geometrically decreasing sequence that bound the diameter of $\mathbf{B}$.*

Proof. From the post-processing step we have

$$\frac{d_i}{d_{i+1}} = \frac{\frac{c_{geo}}{q-p}\left(\frac{d'_i}{\log n}\right)^{\frac{q-p}{p}} d'_i}{\frac{c_{geo}}{q-p}\left(\frac{d'_{i+1}}{\log n}\right)^{\frac{q-p}{p}} d'_{i+1}} = \left(\frac{d'_i}{d'_{i+1}}\right)^{\frac{q}{p}}.$$

Since $\mathbf{d}'$ is a geometrically decreasing sequence and $\frac{q}{p} > 1$, $\mathbf{d}$ is also a geometrically decreasing sequence. As the lengths in each B'_i and d'_i are scaled by the same factor, d_i remains an upper bound for the diameter of B_i for all i. For any edge $e \in B_i$, since $d'_i \geq l'(e) \log n$, we have

$$\begin{aligned} d_i &= \frac{c_{geo}}{q-p}\left(\frac{d'_i}{\log n}\right)^{\frac{q-p}{p}} d'_i \\ &\geq \frac{c_{geo}}{q-p}\left(\frac{d'_i}{\log n}\right)^{\frac{q-p}{p}} l'(e) \log n \\ &= l_{B_i}(e) \log n. \end{aligned}$$

Therefore $\mathbf{d}$ upper bounds the diameters of $\mathbf{B}$ as well.

■

We now check that the union of each level in $\mathbf{B}$ is a subgraph of G in the weighted sense, which makes it an embeddable Bartal decomposition.

Lemma 4.4.2. *For any edge e we have*

$$\sum_i w_{B_i}(e) \leq w(e).$$

Proof. Combining the pre-processing and post-processing steps gives that the total weight of e in all the layers is:

$$\sum_i w_{B_i}(e) = \sum_{\{i | e \in B_i\}} \frac{1}{l_{B_i}(e)} = \frac{p-q}{c_{geo}} \sum_{\{i | e \in B_i\}} \left(\frac{\log n}{d'_i}\right)^{\frac{q-p}{p}} w(e)^{\frac{p}{q}}.$$

Upper bounding the above by $w(e)$ is equivalent to showing

$$\frac{p-q}{c_{geo}} \sum_{\{i | e \in B_i\}} \left(\frac{\log n}{d'_i w(e)^{\frac{p}{q}}}\right)^{\frac{q-p}{p}} \leq 1.$$

Recall that for each i such that $e \in B_i$, we have $d'_i \geq l'(e) \log n$. Substituting $l'(e) = w(e)^{-\frac{p}{q}}$ into this bound on d'_i gives:

$$\frac{\log n}{d'_i w(e)^{\frac{p}{q}}} \leq 1.$$

As d'_is are decreasing geometrically, the following sequence is also decreasing geometrically, with the largest term being at most 1:

$$\left(\left(\frac{\log n}{d'_i w(e)^{\frac{p}{q}}} \right)^{\frac{q-p}{p}} \right)_{\{i: e \in B_i\}}.$$

Applying Fact 4.3.6 with $\epsilon = \frac{q-p}{p}$ we get

$$\frac{p-q}{c_{geo}} \sum_{\{i | e \in B_i\}} \left(\frac{\log n}{d'_i w(e)^{\frac{p}{q}}} \right)^{\frac{q-p}{p}} \leq \frac{p-q}{c_{geo}} \cdot \frac{c_{geo} p}{q-p} \leq 1$$

as we desired.

■

We can also check that the stretch of an edge e with respect to $(\mathbf{B}, \mathbf{d})$ is comparable to its stretch in $(\mathbf{B}', \mathbf{d}')$.

Lemma 4.4.3. *For parameters $0 < p < q < 1$, the ℓ_p-stretch of an edge e in G with respect to $(\mathbf{B}, \mathbf{d})$ and its ℓ_q-stretch in G' with respect to $(\mathbf{B}', \mathbf{d}')$ are related by*

$$str^p_{\mathbf{B},\mathbf{d}}(e) = O\left(\frac{1}{q-p} \log^{p-q} n \cdot str^q_{\mathbf{B}',\mathbf{d}'}(e) \right).$$

Proof. Recall that

$$d_i = \frac{c_{geo}}{q-p} \left(\frac{d'_i}{\log n} \right)^{\frac{q-p}{p}} d'_i = \frac{c_{geo}}{q-p} \cdot \log^{\frac{p-q}{p}} n \cdot d'^{\frac{q}{p}}_i.$$

For an edge cut at level i, we have

$$\begin{aligned} \text{str}_{\mathbf{B},\mathbf{d}}(e) &= \frac{d_i}{l(e)} \\ &= \frac{c_{geo}}{q-p} \cdot \log^{\frac{p-q}{p}} n \cdot \frac{d'^{\frac{q}{p}}_i}{l'(e)^{\frac{q}{p}}} \\ &= \frac{c_{geo}}{q-p} \cdot \log^{\frac{p-q}{p}} n \cdot \left(\text{str}^q_{\mathbf{B}',\mathbf{d}'}(e) \right)^{\frac{1}{p}}. \end{aligned}$$

Taking both sides to the p-th power and using the fact that $p < 1$ gives the desired bound.

■

It is worth noting that when both p and q are bounded away from 1, this procedure is likely optimal up to constants. This is because the best ℓ_p-stretch and ℓ_q-stretch that one could obtain in these settings are $O(\log^p n)$ and $O(\log^q n)$ respectively.

4.4.3 From Decompositions to Trees

It remains to show that an embeddable decomposition can be converted into an embeddable tree. Our conversion routine is based on the laminar-decomposition view of the decomposition. From the bottommost level upwards, we iteratively reduce the interaction of each cluster with other clusters to a single vertex in it, which we term the centers. Centers can be picked arbitrarily, but we require that if a vertex u is a center on level i, it is also a center on all levels $j > i$. Once the centers are picked, we can connect the clusters starting at the bottom level, by connecting all centers of level $i+1$ to the center of the connected component they belong to at level i. This is done by taking the part of B_i involving these centers. We first show that the tree needed to connect them has size at most twice the number of centers.

Lemma 4.4.4. *Given a tree T and a set of k vertices S, there is a tree T_S on $2k-1$ vertices having the set S as its leaves such that:*

- *The distances between vertices in S are the same in T and T_S.*
- *T_S is embeddable into T.*

Proof. The proof is by induction on the number of vertices in T. If T has no more than $2k-1$ vertices, it suffices to set $T_S = T$. For the inductive step suppose the result is true for all trees with at most n vertices, and T has $n+1$ vertices. We will show that there is a tree T' on n vertices that preserves all distances between vertices in S, and is embeddable into T.

If T has a leaf that is not in S, removing that leaf does not affect the distances between the vertices in S and the resulting tree T' is a subgraph and therefore embeddable into T. Otherwise T contains at most k leaves, contributing k to its total degree. Since T contains n edges, the total degree in T is $2n$. Thus the contribution of the $n+1-k$ vertices outside S is at most $2n-k$, so the average degree of vertices outside S at most

$$\frac{2n-k}{n+1-k} = \frac{2-\frac{k}{n}}{1+\frac{1}{n}-\frac{k}{n}} < \frac{2-\frac{k-1}{n}}{1-\frac{k-1}{n}}.$$

Since T does not fall into the base case it can be verified that $\frac{k-1}{n} \leq \frac{1}{2}$, thus the average degree outside S is less than 3, and there must exists a vertex $u \notin S$ of degree 2. Let the two neighbors

of u be v_1 and v_2 respectively. We can then remove u and add an edge between v_1 and v_2 with length $l(u, v_1) + l(u, v_2)$ to preserve the distances within S. This new tree T' is embeddable in T by mapping the edge between v_1 and v_2 to the only path in T between v_1 and v_2.

Since T' has n vertices, the inductive hypothesis gives the existence of a tree T_S on $2k - 1$ vertices preserving distances within S. As T' is embeddable into T, it is easy to check that T_S is embeddable into T as well.

■

Applying this lemma on each cluster in a Bartal decomposition then leads to the overall tree. Pseudocode of this tree construction is given in Algorithm 10.

Algorithm 10 Build-Tree$(G, \mathbf{B})$

Input: weighted graph $G = (V, E, l)$ and a Bartal decomposition $\mathbf{B}$ of G

1: designate a center vertex for each tree in each level of $\mathbf{B}$, such that if a vertex is a center in level i, it is also a center in level $i + 1$
2: merge the copies of a same vertex marked as a center in all the B_is, this produces a single tree T
3: identify the vertices from the last level B_t with the original vertex set V
4: identify the remaining vertices as Steiner vertices
5: apply Lemma 4.4.4 on T and V to produce T_V
6: **return** T_V

Lemma 4.4.5. *Given a graph $G = (V, E, l)$ and an embeddable Bartal decomposition* $\mathbf{B}$, Build-Tree *returns an embeddable tree T_V with $O(n)$ vertices such that for any geometrically decreasing sequence* $\mathbf{d}$ *that bounds the diameters of* $\mathbf{B}$ *and any edge e we have*

$$str_{T_V}(e) = O(str_{\mathbf{B},\mathbf{d}}(e)).$$

Proof. Since $\cup_i B_i$ is embeddable into G, the T obtained from Line 2 of Build-Tree is also embeddable into G. Then in Line 5 we applied Lemma 4.4.4 on T and the vertex set V. Since $|V| = n$, the resulting tree T_V has at most $2n - 1$ vertices and is embeddable into G.

It remains to bound the stretch of edges with respect to T_V. Since an of edge only exists between the non-Steiner vertices, we can bound its stretch in T instead. Consider an edge $e = \{u, v\}$ that is first cut at level i. The tree path between its endpoints goes from u through cluster centers on levels $t, t - 1, \ldots, i$, then back down to v, and the distance traversed on level j is bounded by $2d_j$. As $\mathbf{d}$ is a geometrically decreasing sequence, the total length of this path is bounded by $O(d_i)$.

■

Combining these pieces leads to an algorithm generating low ℓ_p-stretch embeddable trees.

Lemma 4.4.6. *Let $G = (V, E, l)$ be a weighted graph with n vertices, m edges, length function $l : E \to [1, \Delta]$, and parameter $0 < p < 1$. We can construct a distribution over Bartal decompositions such that for any edge e, its expected ℓ_p-stretch in a decomposition sampled from this distribution is $O((\frac{1}{1-p})^2 \log^p n)$.*

Proof. Consider running Embeddable-Decompose with $q = \frac{1+p}{2}$, and Decompose-Simple. By Lemmas 4.3.7 and Lemma 4.4.3, the expected stretch of an edge e in the output decomposition $\mathbf{B}$ with respect to diameter bounds $\mathbf{d}$ is:

$$O\left(\frac{1}{q-p}\log^{p-q} n \cdot \frac{1}{1-q}\log^{q} n\right) = O\left(\left(\frac{1}{1-p}\right)^2 \log^p n\right).$$

Running Build-Tree on this decomposition then gives a tree with the expected stretch on edges are the same. The embeddability of this tree also follows from the embeddability of $\mathbf{B}$ given by Lemma 4.4.2.

To bound the running time, note that as $0 \leq \frac{p}{q} < 1$, the lengths of edges in the preprocessed graph G' are also between 1 and Δ. Both the pre and post processing steps consist of only rescaling edge weights, and therefore take linear time. The total running time then follows from Lemma 4.3.7.

■

4.5 Two-Stage Tree Construction

In this section we give a faster two-stage algorithm for constructing Bartal decompositions. We first quickly build a lower quality decomposition using the same scheme as the AKPW low stretch spanning tree [AKPW95]. Then we proceed in the same way as Bartal's original algorithm and refine the decompositions in a top-down manner. However, with the first stage decomposition, we are able to construct a Bartal decomposition much faster.

Both the AKPW decomposition and the way that our Bartal decomposition routine uses it relies on repeated clustering of vertices. Of course, in an implementation, such clusterings will be represented using various linked-list structures. However, for our analysis, it is helpful to view them as quotient graphs. Given a graph G and a subset of edges A, we define the quotient graph G/A be the graph formed by the connected components of A. Each of connected components of the induced subgraph $G[A]$ becomes a single vertex in G/A, and the edges outside A remains but have their endpoints relabeled accordingly. For our algorithms, it is essential for us to keep multi-edges as separate copies. As a result, all the graphs that we deal with in this section are potentially multi-graphs.

The main advantages offered by the AKPW decomposition are

- It is a bottom-up algorithm that can be performed in linear time.
- Each edge only participates in $O(\log\log n)$ steps of the refinement process in expectation.
- All partition routines are done on graphs with diameter $\text{poly}(\log n)$.

The interaction between the bottom-up AKPW decomposition and the top-down Bartal decomposition leads to some distortions. The rest of this section can be viewed as analyzing this distortion, and the algorithmic gains from having it. We will show that for an appropriately constructed AKPW decomposition, the probability of an edge being cut can be related to a quantity in the ℓ_q norm for some $p < q < 1$. The difference between these two norms then allows us to absorb distortions of size up to $\text{poly}\log n$, and therefore not affecting the quality of the resulting tree. Thus we will work mostly with a different exponent q in this section, and only bring things back to an exponent in p at the very end.

Both the AKPW and the top-down routines will issue multiple calls to Partition. In both cases the granularity of the edge weights will be $\text{poly}(\log n)$. As stated in Section 3, Partition can be implemented in linear time in the RAM model, using the rather involved algorithm presented in [Tho00]. In practice, it is also possible to use the low granularity of edge weights and use Dial's algorithm [Dia69], worsening the total running time of our algorithm to $O(m \log\log n + \log\Delta\,\text{poly}(\log n))$ when all edge lengths are in the range $[1, \Delta]$. Alternatively, we can use the weight-sensitive shortest path algorithm from [KMP11], which works in the pointer machine model, but would be slower by a factor of $O(\log\log\log n)$.

4.5.1 The AKPW Decomposition Routine

We first describe the AKPW algorithm for generating decomposition. The decomposition produced is similar to Bartal decompositions, although we will not impose the strict conditions on diameters in our definition.

Definition 4.5.1. *Let $G = (V, E, l)$ be a connected multi-graph. We say that a sequence of forests* $\mathbf{A}$*, where*

$$\mathbf{A} = (A_0, A_1, \ldots, A_s),$$

is an AKPW decomposition *of G with parameter δ if:*

1. *A_s is a spanning tree of G.*
2. *For any $i < s$, $A_i \subseteq A_{i+1}$.*
3. *The diameter of each connected component in A_i is at most δ^{i+1}.*

Pseudocode for generating this decomposition is given in Algorithm 11. We first bound the diameters of each piece, and the probability of an edge being cut in A_i.

Algorithm 11 AKPW(G, δ)

Input: Weighted multi-graph $G = (V, W, l)$ and a parameter δ.
1: partition E by length such that E_i contains all edges with length within $[\delta^i, \delta^{i+1})$
2: $A_0 \leftarrow \varnothing$
3: $s \leftarrow 0$
4: **while** A_s is not a spanning tree of G **do**
5: $\quad E' \leftarrow E_0 \cup \cdots \cup E_s$
6: $\quad G_s \leftarrow (V, E', \mathbf{1})/A_s$, where $\mathbf{1}$ is the constant function with value 1
7: $\quad T_1, \ldots, T_k \leftarrow$ Partition$(G_s, \delta/3)$
8: $\quad A_{s+1} \leftarrow A_s \cup T_1 \cup \cdots \cup T_k$
9: $\quad s \leftarrow s + 1$
10: **end while**
11: **return** $\mathbf{A} = (A_0, \ldots, A_s)$

Lemma 4.5.2. AKPW(G, δ) *generates with high probability an AKPW decomposition* $\mathbf{A}$ *such that for an edge* $e = \{u, v\}$ *with* $l(e) \in [\delta^i, \delta^{i+1})$ *and some* $j \geq i$*, the probability that* u *and* v *are not connected in* A_j *is at most*

$$\left(\frac{c_P \log n}{\delta}\right)^{j-i},$$

where c_P *is a constant associated with the* Partition *routine. Furthermore, if* $\delta \geq 2c_P \log n$*, this runs in expected* $O(m \log \log^{1/2} n)$ *time in the RAM model.*

Proof. By construction that A_s is a spanning tree, and since A_{i+1} is generated by adding edges to A_i, we have $A_i \subseteq A_{i+1}$. The diameter bounds can be proven by induction on i.

The base case of $i = 0$ follows from the clusters being singletons, and as a result having diameter 0. Now suppose the diameter bounds hold for some level i. Then with high probability each connected component in A_{i+1} corresponds to a (quotient) tree with diameter $\delta/3$ connecting the components in A_i. By definition edges in E_i have length at most δ^{i+1}, and by the inductive hypothesis the diameter of each connected component in A_i is also at most δ^{i+1}. This allows us to bound the diameter of A_{i+1} by

$$\frac{\delta}{3} \cdot \delta^{i+1} + (\frac{\delta}{3} + 1) \cdot \delta^{i+1} \leq \delta^{i+2}.$$

The guarantees of the probabilistic decomposition routine from Lemma 4.3.4 gives us that on any given level, an edge has its two endpoints separated with probability at most $(c_P \log n)/\delta$

for some constant c_P. Since $l(e) \in [\delta^i, \delta^{i+1})$, we have $e \in E_i$. So by the time A_j is constructed, e has gone through $j - i$ rounds of Partition calls, and is present if and only if its endpoints have been separated in each of these steps. Since these are independent events, taking the product of the individual probabilities then gives the claimed bound.

We now analyze the complexity of this algorithm. Notice that time spent in each iteration of the loop is linear in the size of the graph G_s processed in that iteration. If $\delta \geq 2c_P \log n$, then the probability of an edge in E_i appearing in subsequent levels decrease geometrically. This means that the total expected sizes of all the G_ss is $O(m)$. Combining this with the linear running time of Partition gives the expected running time of $O(m)$ once we have bucketed the edges into $E_0, E_1, \ldots$, etc. by length. Under the RAM model of computation, these buckets can be formed in $O(m \log\log^{1/2} n)$ time using the sorting algorithm by Han and Thorup [Han04], which becomes the dominating term in the running time.

■

The expected ℓ_1-stretch bound of any edge can be derived by combining the diameter bounds and cut probabilities of the edges. For an edge on the i-th level, the ratio between its length and the diameter of the jth level can be bounded by δ^{j-i+1}. As j increases, the expected stretch of e then increases by factors of

$$\delta \cdot O\left(\frac{\log n}{\delta}\right) = O\left(\log n\right),$$

which leads to the super-logarithmic bound on the expected ℓ_1-stretch from [AKPW95] With ℓ_p-stretch however, the pth power of the diameter-to-length ratio only increases by factors of δ^p. This means that, as long as the probabilities of an edge being cut increases by factors of less than δ^p, a better bound can be obtained.

4.5.2 Accelerate Bartal's Algorithm using AKPW Decomposition

In this section, we describe how we combine the AKPW decomposition and Bartal's original algorithm into a two-pass algorithm. At a high level, Bartal's algorithm repeatedly partitions the graph in a top-down fashion, and the geometrically decreasing diameters translates to a $O(m \log n)$ running time. The way we achieve a speedup is by contracting vertices that are close to each other, in a way that does not affect the quality of the top-down partition scheme too much. More specifically, we precompute an appropriate AKPW decomposition, and only expose a limited number of layers while running the top-down partition. This way we ensure that each edge only appears in $O(\log\log n)$ calls to the partition routine.

Let $\mathbf{A} = (A_0, A_1, \cdots, A_s)$ be an AKPW decomposition with parameter δ, so that G/A_j is the quotient graph where each vertex corresponds to a cluster of diameter at most δ^{j+1} in the

original graph. In order to partition the graph G into pieces of diameter d, where under some notion d is relatively large compared to δ^{j+1}, we observe that the partition can be done on the quotient graph G/A_j instead. As the complexity of our partition routine is linear in the number of edges, this might bring some potential gain. We use the term *scope* to denote the point at which lower levels of the AKPW decomposition are handled at a coarser granularity. When the top-down algorithm reaches diameter d_i in the diameter sequence $\mathbf{d}$, this cutoff point in the AKPW decomposition is denoted by $scope(i)$, which will be defined later. This two-pass decomposition is formalized in Algorithm 12.

Algorithm 12 DECOMPOSE-TWO-STAGE$(G, \mathbf{d}, \mathbf{A})$

Input: Graph G, diameter sequence $\mathbf{d} = (d_1, \dots, d_t)$, and AKPW decomposition of G, $\mathbf{A} = (A_1, \dots, A_s)$

1: $B_0 \leftarrow A_s$
2: **for** $i = 1, \dots, t$ **do**
3:     if necessary, increase i until $G' = B_{i-1}/A_{scope(i)}$ is non-empty
4:     $B_i \leftarrow \varnothing$
5:     form the length function l_i by increasing all edge lengths in G' to at least $\delta^{scope(i)+1}$
6:     remove all edges with length $d_i / \log n$ or more from G'
7:     **for** each connected component H of G' **do**
8:         $G_1, \dots, G_k \leftarrow$ PARTITION$(H, d_i/3)$
9:         $B_i \leftarrow B_i \cup G_1 \cup \dots \cup G_k$
10:         $B_i \leftarrow B_i \cup A_{scope(i)}$
11:     **end for**
12: **end for**
13: **return** $\mathbf{B} = ((B_1, l_1), \dots, (B_t, l_t))$

We first show that the increase in edge lengths to $\delta^{scope(i)+1}$ still allows us to bound the diameter of the connected components of B_i.

Lemma 4.5.3. *The diameter of each connected component in B_i is bounded by d_i with high probability.*

Proof. By the guarantee of the partition routine, the diameter of each G_i from Line 8 is at most $\frac{d_i}{3}$ with high probability. However, since we are measuring diameter of the components in G, we also need to account for the diameter of the components that were shrunken into vertices when forming G'. These components correspond to connected pieces in $A_{scope(i)}$, therefore the diameters of the corresponding trees are bounded by $\delta^{scope(i)+1}$ with high probability. Line 5 of the algorithm ensures that the length of any edge is more than the diameter of its endpoints. Hence the total increase in diameter from these pieces is at most twice the length of a path in G', and the diameter of these components in G can be bounded by d_i.

∎

Once we established that the diameters of our decomposition is indeed geometrically decreasing, it remains to bound the probability of an edge being cut at each level of the decomposition. In the subsequent sections, we give two different analyses of the algorithm DecomposeTwoStage with different choices of scope. We first present a simple version of our algorithm which ignores a $1/\operatorname{poly}(\log n)$ fraction of the edges, but guarantees an expected ℓ_1-stretch close to $O(\log n)$ for rest of the edges. Then we present a more involved analysis with a careful choice of scope which leads to a tree with small ℓ_p-stretch.

4.5.3 Decompositions that Ignore $1/k$ of the Edges

In this section, we give a simplified algorithm that ignores some fraction of the edges, but guarantees for other edges an expected ℓ_1-stretch of close to $O(\log n)$. We also discuss how this relates to the problem of generating low-stretch subgraphs in parallel and its application to parallel SDD linear system solvers. In this simplified algorithm, we use a naive choice of scope, reaching a small power of $k \log n$ into the AKPW decomposition.

Let $\mathbf{d} = (d_0, d_1, \ldots, d_t)$ be a diameter sequence and let $\mathbf{A} = (A_0, A_1, \ldots, A_s)$ be an AKPW decomposition constructed with parameter $\delta = k \log n$. We let

$$scope(i) = \max\{j \mid \delta^{j+3} \leq d_i\}.$$

Note that $\delta^{scope(i)}$ is always between d_i/δ^4 and d_i/δ^3. We say an edge $e \in E_i$ is *AKPW-cut* if e is cut in A_{i+1}. Furthermore, we say an edge e is *floating* in level i if it exists in $B_{i-1}/A_{scope(i)}$ and has length less than $\delta^{scope(i)+1}$. Note that the floating edges are precisely those edges whose length is increased before running the Bartal decomposition. We say that an edge is *floating-cut* if it is not AKPW-cut, but is cut by the Bartal decomposition at any level in which it is floating.

The simplified analysis in this section will only provide stretch guarantees for edges that are not AKPW-cut or floating-cut. We start by bounding the expected number AKPW-cut edges that are ignored in the analysis.

Lemma 4.5.4. *Let* $\mathbf{A} = AKPW(G, \delta)$ *where* $\delta = k \log n$*. The expected number of AKPW-cut edges in* $\mathbf{A}$ *is at most* $O(\frac{m}{k})$*.*

Proof. For an edge $e \in E_i$, the probability that e is cut in A_{i+1} is at most

$$\frac{c_P \log n}{\delta} = \frac{c_P}{k}$$

by Lemma 4.5.2, where c_P is the constant associated with the partition routine. Linearity of expectation then gives that the expected number of AKPW-cut edges is at most $O(\frac{m}{k})$.

■

We now bound the total number of floating-cut edges.

Lemma 4.5.5. *The expected number of floating-cut edges is $O(\frac{m}{k})$.*

Proof. First, we note that only edges whose length is at least $\frac{d_i}{\delta^4}$ may be floating-cut at level i: any edge of lesser length that is not AKPW-cut will not be contained in $B_{i-1}/A_{scope(i)}$. Furthermore, by the definition of being floating, only edges of lengths at most $\frac{d_i}{\delta^2}$ may be floating. Therefore, an edge of length $l(e)$ may only be floating-cut for levels with $d_i \in [\delta^2 l(e), \delta^4 l(e))$. Since the d_is increase geometrically, there are at most $\log(\delta)$ such levels.

Furthermore, at any given level, the probability that a given edge is floating-cut at the level is at most $O(\frac{\log n}{\delta^2})$, since floating edges are passed to the decomposition with length $\frac{d_i}{\delta^2}$. Taking a union bound over all levels with $d_i \in [\delta^2 l(e), \delta^4 l(e))$, any edge has at most a $O(\frac{\log n \log \delta}{\delta^2})$ probability of being cut. Since $\frac{\log \delta}{\delta} = O(1)$, this is equal to $O(\frac{\log n}{\delta}) = O(\frac{1}{k})$.

Again, applying linearity of expectation implies that the expected number of floating-cut edges is $O(\frac{m}{k})$.

■

Combining these two lemmas, we see that the expected number of ignored edges so far is bounded by $O(\frac{m}{k})$. We can also check that conditioned on an edge being not ignored, its probability of being cut on some level is the same as before.

Lemma 4.5.6. *Let $\mathbf{A} = \text{AKPW}(G, \delta)$. We may associate with the output of the algorithm a set of edges S, with expected size $O(\frac{m}{k})$, such that for any edge e with length $l(e)$, conditioned on $e \notin S$, is cut on the ith level of the Bartal decomposition $\mathbf{B}$ with probability at most*

$$O\left(\frac{l(e)\log n}{d_i}\right).$$

Proof. We let S be the union of the sets of AKPW-cut and floating-cut edges. Fix a level i of the Bartal decomposition: if an edge e that is not AKPW-cut or floating-cut appears in $B_{i-1}/A_{scope(i)}$, then its length is unchanged. If e is removed from G' due to $l(e) \geq d_i/\log n$, the bound becomes trivial. Otherwise, the guarantees of PARTITION then give the cut probability.

■

Lemma 4.5.7. *The simplified algorithm produces with high probability an embeddable Bartal decomposition with diameters bounded by $\mathbf{d}$ where all but (in expectation) $O(\frac{m}{k})$ edges satisfy*

$$\mathbb{E}[str_{\mathbf{B},\mathbf{d}}(e)] \leq O(\log n(\log(k\log n))^2).$$

Proof. Let $p = 1 - 1/\log(k \log n)$ and $q = (1+p)/2$. Applying Lemma 4.5.6 and Lemma 4.3.5 we get that for edges not in S,

$$\mathbb{E}[\mathrm{str}^q_{\mathbf{B},\mathbf{d}}(e)] = O(\log^q n \log(k \log n)).$$

Then using EmbeddableDecompose as a black box we obtain an embeddable decomposition with expected l_p-stretches of $O(\log^p n(\log(k \log n))^2)$ for non-removed edges.

By repeatedly running this algorithm, in an expected constant number of iterations, we obtain an embeddable decomposition $\mathbf{B}$ with diameters bounded by $\mathbf{d}$ such that for a set of edges $E' \subseteq E$ and $|E'| \geq m - O(\frac{m}{k})$,

$$\sum_{e \in E'} \mathbb{E}[\mathrm{str}^q_{\mathbf{B},\mathbf{d}}(e)] = O(m \log^q n(\log(k \log n))^2).$$

By Markov's inequality, at most $1/k$ of the edges in E' can have

$$\mathrm{str}^q_{\mathbf{B},\mathbf{d}}(e) \geq O(k \log^q n(\log(k \log n))^2).$$

This gives another set of edges E'' with size at least $m - O(\frac{m}{k})$ such that any edge $e \in E''$ satisfies

$$\mathrm{str}^q_{\mathbf{B},\mathbf{d}}(e) \leq O(k \log^q n(\log(k \log n))^2) \leq O((k \log n)^2).$$

But for each of these edges

$$\begin{aligned}
\mathrm{str}_{\mathbf{B},\mathbf{d}}(e) &= (\mathrm{str}^q_{\mathbf{B},\mathbf{d}}(e))^{1/q} \\
&\leq (\mathrm{str}^q_{\mathbf{B},\mathbf{d}}(e))^{1+2/\log(k \log n)} \\
&\leq \mathrm{str}^q_{\mathbf{B},\mathbf{d}}(e) \cdot O\left((k \log n)^{4/\log(k \log n)}\right) \\
&= O(\mathrm{str}^q_{\mathbf{B},\mathbf{d}}(e)).
\end{aligned}$$

Excluding these high-stretch edges, the ℓ_1 stretch is thus at most a constant factor worse than the ℓ_q stretch, and can be bounded by $O(\log n(\log(k \log n))^2)$.

■

The total running time of DecomposeTwoStage is dominated by the calls to Partition. The total cost of these calls can be bounded by the expected number of calls that an edge participates in.

Lemma 4.5.8. *For any edge e, the expected number of iterations in which e appears is bounded by $O(\log(k \log n))$.*

Proof. As pointed out in the proof of 4.5.5, an edge that is not AKPW-cut only appears in level i of the Bartal decomposition if $l(e) \in [\frac{d_i}{\delta^5}, \frac{d_i}{\log n})$. Since the diameters decrease geometrically, there are at most $O(\log(k \log n))$ such levels. AKPW-cut edges can appear sooner than other edges from the same weight bucket, but using an argument similar to the proof of Lemma 4.5.4 we observe that the edge propagates up j levels in the AKPW decomposition with probability at most $(\frac{1}{k})^j$. Therefore the expected number of such appearances by an APKW-cut edge is at most $\sum_i (\frac{1}{k})^i = O(1)$.

■

Combining all of the above we obtain the following result about our simplified algorithm. The complete analysis of its running time is deferred to Section 4.5.5.

Lemma 4.5.9. *For any k, given an AKPW decomposition $\mathbf{A}$ with $\delta = k \log n$, we can find in $O(m \log(k \log n))$ time an embeddable Bartal decomposition such that for all but expected $O(\frac{m}{k})$ edges have expected total ℓ_1-stretch of at most $O(m \log n (\log(k \log n))^2)$.*

Parallelization

If we relax the requirement of asking for a tree, the above analysis shows that we can obtain low stretch subgraphs with an expected stretch of $O(\log n (\log(k \log n))^2)$ for all but $O(\frac{m}{k})$ edges. As our algorithmic primitive Partition is parallel parallelization, we also obtain a parallel algorithm for constructing low stretch subgraphs. These subgraphs are used in the parallel SDD linear system solver by [BGK+14]. By observing that Partition is run on graphs with edge weights within δ of each other and hop diameter at most polynomial in $\delta = k \log n$, and using the parallel tree-contraction routines [MR89] to extract the final tree, we can obtain the following result.

Lemma 4.5.10. *For any graph G with polynomially bounded edge weights and $k = O(\text{poly}(\log n))$, in $O(k \log^2 n \log \log n)$ depth and $O(m \log n)$ work we can generate an embeddable tree of size $O(n)$ such that the total ℓ_1-stretch of all but $O(\frac{m}{k})$ edges of G is $O(m \log n (\log(k \log n))^2)$.*

4.5.4 Bounding Expected ℓ_p-Stretch of Any Edge

In this section we present our full algorithm and bound the expected ℓ_p-stretch of all the edges. Unlike in the previous section, we can no longer ignore edges whose lengths we increase while performing the top-down partition, we need to choose the scope carefully in order to control their probability of being cut during the second stage of the algorithm. We start off by choosing a different δ when computing the AKPW decomposition.

Lemma 4.5.11. *If* $\mathbf{A}$ *is generated by a call to* AKPW(G, δ) *with*

$$\delta \geq (c_P \log n)^{\frac{1}{1-q}},$$

then the probability of an edge $e \in E_i$ *being cut in level* j *is at most* $\delta^{-q(j-i)}$.

Proof. Manipulating the condition on δ gives that $c_P \log n \leq \delta^{1-q}$, and therefore using Lemma 4.5.2 we can bound the probability by

$$\left(\frac{c_P \log n}{\delta}\right)^{j-i} \leq \left(\frac{\delta^{1-q}}{\delta}\right)^{j-i} = \delta^{-q(j-i)}.$$

■

Since δ is poly$(\log n)$, we can use this bound to show that expected ℓ_p-stretch of an edge in an AKPW-decomposition can be bounded by poly$(\log n)$. The exponent here can be optimized by taking into account the trade-offs given in Lemma 4.3.5.

This extra factor of δ can also be absorbed into the analysis of Bartal decompositions. When the length $l(e)$ of an edge e is significantly less than d, the partitioning diameter, the difference between $\frac{l(e)\log n}{d}$ and $\left(\frac{l(e)\log n}{d}\right)^q$ is more than δ. This means that for a floating edge that originated much lower in the AKPW decomposition, we can afford to increase its probability of being cut by a factor of δ.

From the perspective of the low-diameter decomposition routine, this step corresponds to increasing the length of an edge. This increase in length can then be used to bound the diameter of a cluster in the Bartal decomposition, and also ensures that all edges that we consider have lengths close to the diameter that we partition into. On the other hand, in order to control this increase in lengths, and in turn to control the increase in the cut probabilities, we need to use a different scope when performing the top-down decomposition.

Definition 4.5.12. *For an exponent* q *and a parameter* $\delta \geq \log n$*, we let the scope of a diameter sequence* $\mathbf{d}$ *be*

$$scope(i) \stackrel{def}{=} \max_i \left\{\delta^{i+\frac{1}{1-q}+1} \leq d_i\right\}.$$

Note that for small d, $scope(i)$ may be negative. As we will refer to $A_{scope(i)}$, we assume that $A_i = \emptyset$ for $i < 0$. Our full algorithm can then be viewed as only processing the edges within the scope using Bartal's top-down algorithm. Its pseudocode is given in Algorithm 12.

Note that it is not necessary to perform explicit contraction and expansion of the AKPW clusters in every recursive call. In an effective implementation, they can be expanded gradually, as $scope(i)$ is monotonic in d_i.

The increase in edge lengths leads to increases in the probabilities of edges being cut. We next show that because the AKPW decomposition is computed using a higher norm, this increase can be absorbed, giving a probability that is still closely related to the pth power of the ratio between the current diameter and the length of the edge.

Lemma 4.5.13. *Assume* $\mathbf{A} = \text{AKPW}(G, \delta)$ *with parameter specified as above. For any edge e with length $l(e)$ and any level i, the probability that e is cut at level i of* $\mathbf{B} = \textsc{DecomposeTwoStage}(G, \mathbf{d}, \mathbf{A})$ *is*

$$O\left(\left(\frac{l(e)\log n}{d_i}\right)^q\right).$$

Proof. There are two cases to consider based whether the length of the edge is more than $\delta^{scope(i)+1}$. If it is and it appears in G', then its length is retained. The guarantees of Partition then gives that it is cut with probability

$$O\left(\frac{l(e)\log n}{d_i}\right) \leq O\left(\left(\frac{l(e)\log n}{d_i}\right)^q\right),$$

where the inequality follows from $l(e)\log n \leq d_i$.

Otherwise, since we contracted the connected components in $A_{scope(i)}$, the edge is only cut at level i if it is both cut in $A_{scope(i)}$ and cut by the partition routine. By Lemma 4.5.11, if the edge is from E_j, its probability of being cut in $A_{scope(i)}$ can be bounded by $\delta^{-q(scope(i)-j)}$. Combining this with the fact that $\delta^j \leq l(e)$ allows us to bound this probability by

$$\left(\frac{l(e)}{\delta^{scope(i)}}\right)^q.$$

Also, since the weight of the edge is set to $\delta^{scope(i)+1}$ in G', its probability of being cut by Partition is

$$O\left(\frac{\delta^{scope(i)+1}\log n}{d_i}\right).$$

As the partition routine is independent of the AKPW decomposition routine, the overall probability can be bounded by

$$O\left(\frac{\delta^{scope(i)+1}\log n}{d_i} \cdot \left(\frac{l(e)}{\delta^{scope(i)}}\right)^q\right) = O\left(\left(\frac{l(e)\log n}{d_i}\right)^q \cdot \delta \log^{1-q} n \cdot \left(\frac{\delta^{scope(i)}}{d_i}\right)^{1-q}\right).$$

Recall from Definition 4.5.12 that $scope(i)$ is chosen to satisfy $\delta^{scope(i)+\frac{1}{1-q}+1} \leq d_i$. This along with the assumption that $\delta \geq \log n$ gives

$$\delta \log^{1-q} n \cdot \left(\frac{\delta^{scope(i)}}{d_i}\right)^{1-q} \leq \delta^{2-q}\left(\delta^{-\frac{2-q}{1-q}}\right)^{1-q} \leq 1.$$

Therefore, in this case the probability of e being cut can also be bounded by $O\left(\left(\frac{l(e)\log n}{d_i}\right)^q\right)$.

■

Combining this bound with Lemma 4.3.5 and setting $q = \frac{1+p}{2}$ gives the bound on ℓ_p-stretch.

Corollary 4.5.14. *If q is set to $\frac{1+p}{2}$, we have for any edge e*

$$\mathbb{E}[str_{\mathbf{B},\mathbf{d}}(e)] \leq O\left(\frac{1}{1-p}\log^p n\right).$$

Therefore, we can still obtain the properties of a good Bartal decomposition by only considering edges in the scope during the top-down partition process. On the other hand, this shrinking drastically improves the performance of our algorithm.

Lemma 4.5.15. *Assume $\mathbf{A} = \text{AKPW}(G, \delta)$. For any edge e, the expected number of iterations of* DecomposeTwoStage *in which e is included in the graph given to* Partition *can be bounded by $O(\frac{1}{1-p}\log\log n)$.*

Proof. Note that for any level i it holds that

$$\delta^{scope(i)} \geq d_i \delta^{-\frac{1}{1-q}-2}.$$

Since the diameters of the levels decrease geometrically, there are at most $O(\frac{1}{1-q}\log\log n)$ level is with $l(e) \in [d_i\delta^{-\frac{1}{1-q}-2}, \frac{d_i}{\log n})$.

The expected number of occurrences of e in lower levels can be bounded using Lemma 4.5.11 in a way similar to the proof of the above Lemma. Summing over all the levels i where e is in a lower level gives:

$$\sum_{i: l(e) < d_i \delta^{-\frac{1}{1-q}-2}} \left(\frac{l(e)}{\delta^{scope(i)}}\right)^q.$$

Substituting in the bound on $\delta^{scope(i)}$ from above and rearranging terms we get the following upper bound:

$$\leq \sum_{i: l(e) \leq d_i \delta^{-\frac{1}{1-q}-2}} \left(\frac{l(e)}{d_i}\delta^{\frac{1}{1-q}+2}\right)^q.$$

As the d_is increase geometrically, this is a geometric sum with the first term being at most 1. Therefore the expected number of times that e appears on some level i while being out of scope is $O(1)$.

■

Recall that each call to Partition runs in time linear in the number of edges. This then implies a total cost of $O(m \log\log n)$ for all the partition steps. We can now proceed to extract a tree from this decomposition, and analyze the overall running time.

4.5.5 Returning a Tree

We now give the overall algorithm and analyze its performance. Introducing the notion of scope in the recursive algorithm limits each edge to appear in at most $O(\log\log n)$ levels. Since each of these calls to Partition runs in time linear in the input size, this should give a total run time of $O(m \log\log n)$. However, the goal of the algorithm as stated is to produce a Bartal decomposition, which has a spanning tree at each level. Explicitly generating this gives a total size of $\Omega(nt)$, where t is the number of recursive calls. As a result, we will circumvent this by storing only an implicit representation of the Bartal decomposition to find the final tree.

This smaller implicit representation stems from the observation that large parts of the B_is are trees from the AKPW decomposition. As a result, such succinct representations are possible if we have pointers to the connected components of A_i. We first analyze the quality and size of this implicit decomposition, and the running time for producing it.

Algorithm 13 Decompose(G, p)

Input: Graph G and stretch exponent parameter p.

1: $q \leftarrow (1+p)/2$
2: $\delta \leftarrow (c \log n)^{\frac{1}{q-p}}$
3: $\mathbf{A}, \mathbf{d} \leftarrow \text{AKPW}(G, \delta)$
4: $\mathbf{B} \leftarrow$ Decompose-Two-Stage$(G, \mathbf{d}, \mathbf{A})$
5: **return** $\mathbf{B}, \mathbf{d}$

Lemma 4.5.16. *There is a routine that for any graph G and parameter $p < 1$, produces in expected $O(\frac{1}{1-p} m \log\log n)$ time an implicit representation of a Bartal decomposition $\mathbf{B}$ with expected size $O(\frac{1}{1-p} m \log\log n)$ and diameter bounds $\mathbf{d}$ such that with high probability:*

1. $\mathbf{B}$ *is embeddable into G.*

2. *For any edge e,*

$$\mathbb{E}[str^{p}_{\mathbf{B},\mathbf{d}}(e)] \leq O\left((\frac{1}{1-p})^2 \log^p n \right).$$

3. $\mathbf{B}$ *consist of edges and weighted connected components of an AKPW decomposition.*

Proof. Consider calling Embeddable-Decompose from Section 4.4.2 with the routine given in Algorithm 13. The properties of **B** and the bounds on stretch follows from Lemma 4.4.3 and Corollary 4.5.14.

Since the number of AKPW components implicitly referred to at each level of the recursive call is bounded by the total number of vertices, and in turn the number of edges, the total number of such references is bounded by the size of the G's as well. This gives the bound on the size of the implicit representation.

We now bound the running time. In the RAM model, bucketing the edges and computing the AKPW decomposition can be done in $O(m \log \log n)$ time. The resulting tree can be viewed as a laminar decomposition of the graph. This is crucial for making the adjustment in DecomposeTwoStage in $O(1)$ time to ensure that $A_{scope(i)}$ is disconnected. As we set q to $\frac{1+p}{2}$, by Lemma 4.5.15, each edge is expected to participate in $O(\frac{1}{1-p} \log \log n)$ recursive calls, which gives a bound on the expected total.

The transformation of the edge weights consists of a linear-time pre-processing, and scaling each level by a fixed parameter in the post-post processing step. This process affects the implicit decomposition by changing the weights of the AKPW pieces, which is can be done implicitly in $O(1)$ time by attaching extra 'flags' to the clusters.

■

It remains to show that an embeddable tree can be generated efficiently from this implicit representation. We define the notion of a contracted tree with respect to a subset of vertices, obtained by repeating the two combinatorial steps that preserve embeddability described in Section 4.2.

Definition 4.5.17. *We define the* contraction *of a tree T to a subset of its vertices S as the unique tree arising from repeating the following operations while possible:*

1. *Removal of a degree 1 vertex not in S.*
2. *Contraction of a degree 2 vertex not in S.*

We note that it is enough to find contractions of the trees from the AKPW decomposition to the corresponding sets of connecting endpoints in the implicit representation. Here we use the fact that the AKPW decomposition is in fact a single tree.

Fact 4.5.18. *Let $\mathbf{A} = (A_0, \ldots, A_s)$ be an AKPW decomposition of G. Let S be a subset of vertices of G. For any $i \in \{0, \ldots, s\}$, if S is contained in a single connected component of A_i, then the contraction of A_i to S is equal to the contraction of A_s to S.*

This allows us to use data structures to find the contractions of the AKPW trees to the respective vertex sets more efficiently.

Lemma 4.5.19. *Given a tree A_s on the vertex set V (with $|V| = n$) and subsets $S_1, \ldots, S_k$ of V where $\sum_i |S_i| = O(n)$, we can generate the contractions of A_s to each of the sets S_i in time $O(n)$ in the RAM model and $O(n\alpha(n))$ in the pointer machine model.*

Proof. Root A_s arbitrarily. Note that the only explicit vertices required in the contraction of A_s to a set $S \subseteq V$ are

$$\Gamma(S) \stackrel{\text{def}}{=} S \cup \{LCA(u,v) \mid u, v \in S\}$$

where $LCA(u,v)$ denotes the lowest common ancestor of u and v in A_s. Moreover, it is easily verified that if we sort the vertices $v_1, \ldots, v_{|S|}$ of S according to the depth first search pre-ordering, then

$$\Gamma(S) = S \cup \{LCA(v_i, v_{i+1}) \mid 1 \leq i < |S|\}.$$

We can therefore find $\Gamma(S_i)$ for each i simultaneously in the following steps.

1. Sort the elements of each S_i according to the pre-ordering, using a single depth-first traversal of A_s.
2. Prepare a list of lowest common ancestor queries for each pair of vertices adjacent in the sorted order in each set S_i.
3. Answer all the queries simultaneously using an off-line lowest common ancestor finding algorithm.

Since the total number of queries in the last step is $O(n)$, its running time is $O(n\alpha(n))$ in the pointer machine model using disjoint union [Tar79], and $O(n)$ in the RAM model [GT83].

Once we find the sets $\Gamma(S_i)$ for each i, we can reconstruct the contractions of A_s as follows.

1. Find the full traversal of the vertices in $\Gamma(S_i)$ for each i, using a single depth first search traversal of A_s.
2. Use this information to reconstruct the trees [Vui80].

■

Applying this procedure to the implicit decomposition then leads to the final embeddable tree.

Proof of Theorem 4.1.1. Consider the distribution over Bartal decompositions given by Lemma 4.5.16. We will apply the construction given in Lemma 4.4.5, albeit in a highly efficient manner.

For the parts of the decomposition that are explicitly given, the routine runs in linear time. The more intricate part is to extract the smaller contractions from the AKPW components

that are referenced to implicitly. Since all levels of the AKPW decomposition are subtrees of A_s, these are equivalent to finding contractions of A_s for several sets of vertices, as stated in Fact 4.5.18. The algorithm given in Lemma 4.5.19 performs this operation in linear time. Concatenating these trees with the one generated from the explicit part of the decomposition gives the final result.

■

4.6 Sufficiency of Embeddability

In the construction of our trees, we made a crucial relaxation of only requiring our tree to be embeddable, rather than restricting it to be a subgraph. In this section, we show that linear operators on the resulting graph can be related to linear operators on the original graph. Our analysis is applicable to ℓ_∞ flows as well.

The spectral approximation of two graphs can be defined in terms of their Laplacian matrices. For matrices, we can define a partial ordering $\preceq$ where $\mathbf{A} \preceq \mathbf{B}$ if $\mathbf{B} - \mathbf{A}$ is positive semidefinite. That is, for any vector $\mathbf{x}$ we have

$$\mathbf{x}^T \mathbf{A} \mathbf{x} \leq \mathbf{x}^T \mathbf{B} \mathbf{x}.$$

If we let H be the graph formed by adding the tree to G, then our goal is to bound L_G and L_H with each other. Instead of doing this directly, it is easier to relate their pseudoinverses. This will be done by interpreting $\mathbf{x}^T L^\dagger \mathbf{x}$ in terms of the energy of electrical flows. The energy of an electrical flow is defined as the sum of squares of the flows on the edges multiplied by their resistances, which in our case are equal to the lengths of the edges. Given a flow $f \in \mathbb{R}^E$, we will denote its electrical energy using

$$\mathcal{E}_G(f) \stackrel{\text{def}}{=} \sum_e l(e) f_e^2.$$

The residue of a flow f is the net in/out flow at each vertex. This give a vector on all vertices, and finding the minimum energy of flows that meet a given residue is equivalent to computing $\mathbf{x}^T L^\dagger \mathbf{x}$. The following fact plays a central role in the monograph by Doyle and Snell [DS84]:

Fact 4.6.1. *Let G be a connected graph. For any vector $\mathbf{x}$ orthogonal to the all-ones vector, $\mathbf{x}^T L_G^\dagger \mathbf{x}$ equals the minimum electrical energy of a flow with residue $\mathbf{x}$.*

Lemma 4.6.2. *Let $G = (V_G, E_G, w_G)$ and $H = (V_H, E_H, w_H)$ be graphs such that G is a subgraph of H in the weighted sense and $H \setminus G$ is embeddable in G. Furthermore, let the graph Laplacians of G and H be L_G and L_H respectively. Also, let Π be the $|V_G| \times |V_H|$ matrix with one 1 in each row at the position*

that vertex corresponds to in H and 0 everywhere else, and Π_1 the orthogonal projection operator onto the part of $\Re^{V_G}$ that's orthogonal to the all-ones vector. Then we have:

$$\frac{1}{2}L_G^{\dagger} \preceq \Pi_1 \Pi L_H^{\dagger} \Pi^T \Pi_1^T \preceq L_G^{\dagger}.$$

Proof. Since $\Pi_1^T = \Pi_1$ projects out any part space spanned by the all-ones vector, and is this precisely the null space of L_G, it suffices to show the result for all vectors $\mathbf{x}_G$ orthogonal to the all-1s vector. These vectors are in turn valid demand vectors for electrical flows. Therefore, the statement is equivalent to relating the minimum energies of electrical flows routing $\mathbf{x}_G$ on G and $\Pi^T \mathbf{x}_G$ on H.

We first show that flows on H take less energy than the ones in G. Let $\mathbf{x}_G$ be any vector orthogonal to the all-ones vector, and f_G^* be the flow of minimum energy in G that meets demand $\mathbf{x}_G$. Setting the same flow on the edges of $E(G)$ in H and 0 on all other edges yields a flow f_H. The residue of this flow is the same residue in V_G, and 0 everywhere else, and therefore is equal to $\Pi^T \mathbf{x}_G$. Since G is a subgraph of H in the weighted sense, the lengths of these edges can only be less. Therefore the energy of f_H is at most the energy of f_G and we have

$$\mathbf{x}_G^T \Pi L_H^{\dagger} \Pi^T \mathbf{x}_G \leq \mathcal{E}_H(f_H) \leq \mathcal{E}_G(f_G^*) = \mathbf{x}_G^T L_G^{\dagger} \mathbf{x}_G.$$

For the reverse direction, we use the embedding of $H \setminus G$ into G to transfer the flow from H into G. Let $\mathbf{x}_G$ be any vector orthogonal to the all-ones vector, and f_H^* the flow of minimum energy in H that has residue $\Pi^T \mathbf{x}_G$. This flow can be transformed into one in G that has residue $\mathbf{x}_G$ using the embedding. Let vertex/edge mapping of this embedding be π_V and π_E respectively.

If an edge $e \in E_H$ is also in E_G, we keep its flow value in G. Otherwise, we route its flow along the path that the edge is mapped to. Formally, if the edge is from u to v, $f_H(e)$ units of flow is routed from $\pi_V(u)$ to $\pi_V(v)$ along $path(e)$. We first check that the resulting flow, f_G has residue $\mathbf{x}_G$. The net amount of flow into a vertex $u \in V_G$ is

$$\begin{aligned}
&\sum_{uv \in E_G} f_H^*(e) + \sum_{u'v' \in E_H \setminus E_G, \pi_V(u') = u} f_H^*(e) \\
&= \sum_{uv \in E_G} f_H^*(e) + \sum_{u' \in V_H, \pi_V(u') = u} \left(\sum_{u'v' \in E_H \setminus E_G} f_H^*(e) \right) \\
&= \sum_{u' \in V_H, \pi_V(u') = u} \sum_{u'v' \in E_H} f_H(e) \\
&= \sum_{u' \in V_H, \pi_V(u') = u} \left(\Pi^T \mathbf{x}_G \right)(e) \\
&= x_G(u).
\end{aligned}$$

Reordering the summations and noting that $\Pi(u) = u$ gives the second equality. The last equality follows from the fact that $\pi_V(u) = u$, and all vertices not in V_G having residue 0 in $\Pi^T \mathbf{x}_G$.

To bound the energy of this flow, the property of the embedding gives that if split the edges of G into the paths that form the embedding, each edge is used at most once. Therefore, if we double the weights of G, we can use one copy to support G, and one copy to support the embedding. The energy of this flow is then the same. Hence there is an electrical flow f_G in G such that $\mathcal{E}_G(f_G) \leq 2\mathcal{E}_H(f_H^*)$. Fact 4.6.1 then gives that it is an upper bound for $\mathbf{x}_G^T L_G^{\dagger} \mathbf{x}_G$, completing the proof.

∎

Chapter 5

Parallel Shortest Paths and Hopsets

5.1 Introduction

In this Chapter we discuss another application of our low diameter decomposition algorithm: approximating shortest paths in parallel. Given a weighted graph G and vertices s and t, the st-shortest path problem looks for a path in G from s to t that minimizes the total length of edges on the path. Algorithms for finding shortest paths have been studied extensively. In the sequential setting, highly efficient algorithms are known when all lengths are non-negative. These algorithms are based on Dijkstra's algorithm [Dij59], which in turn can be viewed as a generalization of breadth-first search: one explores vertices in order of their distances to the starting vertex s. In the standard comparison-addition model, the Fibonacci heap by Fredman and Tarjan [FT87] allows Dijkstra's algorithm to run in $O(m + n \log n)$ time. In undirected graphs where edge lengths are integers from $[1, L]$, Pettie and Ramachandran gave an algorithm that runs in $O(m + n \log \log L)$ time [PR05]. Further speedups are also possible in the RAM model, where a linear time algorithm was given by Thorup [Tho00].

One of the shortcomings of these algorithms is that the underlying greedy procedure can have long chains of sequential dependencies. Processing vertices in increasing order of distances means that a vertex is dependent on its predecessor along the shortest path from the source. This makes it difficult for parallel algorithms to obtain speedups over sequential algorithms using a modest number of processors, especially on sparse graphs. One standard way to measure the complexity of a parallel algorithm is to measure two parameters, its depth and work. The depth of a parallel algorithm is the length of its longest sequential dependency, and is often referred to as parallel time, since it is the running time of the algorithm assuming an unlimited number of processors. The work of a parallel algorithm is the total number of operation performed by all the processors.

However, a bottleneck that is often more important in practice is work divided by the number of processors. The simplest parallel algorithm for shortest path is based on matrix multiplication, which takes $O(n^3)$ work and $O(\text{poly}\log n)$ depth. This means on a sparse graph, $O(n^2)$ processors are required to obtain speedups over Dijkstra's algorithm. Thus from a practical standpoint, we would like parallel algorithms that achieve low depth (polylogarithmic or even simply sublinear) with about the same amount of work as their sequential counterpart (i.e. nearly linear in the number of edges). However such an algorithm remained elusive for decades, and this is referred to as the *transitive closure bottleneck* in the literature.

A natural direction is then to consider approximations. Hopsets were formalized by Cohen [Coh00] as a crucial component for parallel approximate shortest paths algorithms, and was implicitly used in a number of earlier works [KS97, UY91]. The goal is to add a set of extra edges to the graph so that we can get good approximation on distances by only considering paths with few edges.

Definition 5.1.1. *Given a graph $G = (V, E, l)$, a (ϵ, h, m')-hopset is a set of edges E' such that:*

1. $|E'| \leq m'$.
2. *Each edge $\{u, v\} \in E'$ corresponds to a uv-path p in G such that $l(u, v) = l(p)$.*
3. *For any vertices u and v, with probability $1/2$ we have:*

$$dist^h_{E \cup E'}(u, v) \leq (1 + \epsilon) dist_E(u, v).$$

Here $dist^h_{E \cup E'}(\cdot, \cdot)$ denotes the shortest path distance using only h edges from E and E'.

Given an (ϵ, h, m')-hopset, Klein and Subramanian[KS97] showed how to approximate shortest paths in $O(h \log^* n/\epsilon)$ depth and $O((m + m')/\epsilon)$, where m is the number of edges including the additional hopset. Thus in the rest of this section, we focus on the problem of finding hopsets with $h = o(n)$ and $m' = O(m)$.

Building on Cohen's work [Coh00] and using the low diameter graph decomposition from Chapter 2, we proved the following result in [MPVX15].

Theorem 5.1.2. *There exists an algorithm that given as input an undirected graph G with non-negative edge lengths, and parameters $\alpha, \epsilon \in (0, 1)$, preprocesses the graph in $O(m\epsilon^{-2-\alpha} \log^{3+\alpha} n)$ work and $O\left(n^{\frac{4+\alpha}{4+2\alpha}} \epsilon^{-1-\alpha} \log^2 n \log^* n\right)$ depth, so that for any vertices s and t one can find a $(1 + \epsilon)$-approximation to the $s - t$ shortest path in $O(m\epsilon^{-1-\alpha})$ work and $O\left(n^{\frac{4+\alpha}{4+2\alpha}} \epsilon^{-2-\alpha}\right)$ depth.*

Figure 5.1 contains a comparison of our hopset construction and previous works (ignoring the dependencies on ϵ). Our main contribution here is to achieve nearly linear work with sublinear depth, as oftentimes work is the bottleneck to empirical performances due to limited number of processors. For Cohen's algorithm, $\Omega(n^\alpha)$ processors are needed for parallel

Hop count	Size	Work	Depth	Notes
$O(n^{1/2})$	$O(n)$	$O(mn^{0.5})$	$O(n^{0.5}\log n)$	[KS97, SS99]
$O(\text{poly}\log n)$	$O(n^{1+\alpha}\text{poly}\log n)$	$\tilde{O}(mn^{\alpha})$	$O(\text{polylog}\, n)$	[Cohoo]
$(\log n)^{O((\log\log n)^2)}$	$O\left(n^{O(\frac{1}{\log\log n})}\right)$	$\tilde{O}\left(mn^{O(\frac{1}{\log\log n})}\right)$	$(\log n)^{O((\log\log n)^2)}$	[Cohoo]
$O(n^{\frac{4+\alpha}{4+2\alpha}})$	$O(n)$	$O(m\log^{3+\alpha} n)$	$O(n^{\frac{4+\alpha}{4+2\alpha}})$	new

Figure 5.1: Performances of Hopset Constructions, omitting ϵ dependency.

speedups in both the construction and query stages [1]. In our case, if ϵ is a constant, $O(\log^{3+\alpha} n)$ processors are sufficient to achieve parallel speedups. Furthermore, once a hopset is constructed, even a constant number of processors suffices for parallel speedups when querying.

5.2 Hopset Construction

5.2.1 Hopsets in Unweighted Graphs

Our hopset construction is based on a recursive application of the exponential start time clustering from Chapter 2. We will designate some of the clusters produced, specifically the larger ones, as special. Since each vertex belongs to at most one cluster, there cannot be too many large clusters. As a result we can afford to compute distances from their centers to all other vertices, and keep a subset of them as hopset edges in the graph. There are two kinds of edges that we keep:

1. *Star edges* between the center of a large cluster and all other vertices in that cluster.
2. *Clique edges* between the all the centers of large clusters in one level of the hierarchy.

In other words, in building the hopset we put a star on top of each large cluster and connect their centers into a clique. Then if an optimal *s*-*t* path p^* encounters two or more of these large clusters, we can jump from the first to the last by going through their centers. One possible interaction between the decomposition scheme and a path p^* in one level of the algorithm is shown in Figure 5.2.

This allows us to replace what hopefully is a large part of p^* with only three edges: two star edges and one clique edge. However this may increase the length of the path by an amount roughly equal to the diameter of the large clusters. But as this distortion can only happen once, it is acceptable as long as the diameter of the clusters are less than $\epsilon \cdot l(p^*)$. Our algorithm then recursively builds hopsets on the small clusters. The exponential start time clustering

[1] A more detailed analysis leads to a tighter bound of $\Omega(\exp(\sqrt{\log n}))$

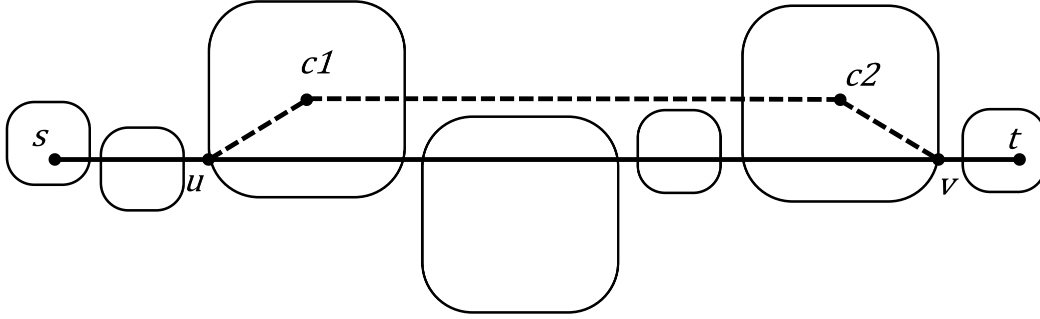


Figure 5.2: Interaction of a shortest path with the decomposition scheme. Hop set edges connecting the centers of large clusters allow us to "jump" from the first vertex in a large cluster (u), to the last vertex of a large cluster (v). The edges $\{u, c_1\}$, $\{c_2, v\}$ are star edges, while $\{c_1, c_2\}$ is a clique edge.

algorithm guarantees that p^* does not interact with too many such clusters, so once again we can afford a reasonable distortion within each of them.

Formally, two parameters control the behavior of our algorithm: the parameter β with which the clustering routine is run, and the threshold ρ by which a cluster is deemed large. The algorithm then has the following main steps:

1. Compute an exponential start time clustering with parameter β
2. Identify clusters with more than n/ρ vertices as large clusters.
3. Construct star and clique edges from the centers of each large cluster.
4. Recurse on the small clusters.

Our choice of β at each level of the recursion is constrained by the additive distortion that we can incur. Consider a cluster obtained at the ith level of the decomposition ran with parameter β_i. Since the path has length d and each edge is cut with probability β_i, the path is expected to be broken into $\beta_i d$ pieces. Therefore on average, the length of each piece in a cluster is about β_i^{-1}. The diameter of a cluster in the next level on the other hand can be bounded by $k\beta_{i+1}^{-1} \log n$, where the constant $k \geq 1$ can be chosen to achieved the desired success probability using Lemma 2.2.2. Therefore, we need to set β_{i+1} so that:

$$k\beta_{i+1}^{-1} \log n \leq \epsilon \beta_i^{-1}$$
$$\beta_{i+1} \geq \left(k\epsilon^{-1} \log n\right) \beta_i.$$

In other words, the βs need to increase from one level to the next by a factor of $k\epsilon^{-1} \log n$ where $\epsilon < 1$ the distortion parameter. This means that the path p^* is cut with granularity that increases by a factor of $O(\epsilon^{-1} \log n)$ each time. Note that the number of edges cut in all levels of

the recursion serves as a rough estimate to the number of hops in our shortcut path. Therefore, a different termination condition is required to ensure that the path is not completely shattered by the decomposition scheme. As we only recurse on small clusters, if we require their size to decrease at a much faster rate than the increase in β, our recursion will terminate with most pieces of the path within large clusters. To achieve this, we introduce a parameter ρ to control this rate of decrease. Given a cluster with n vertices, we designate a cluster X_i to be small if $|X_i| \leq n/\rho$. As our goal is a faster rate of decrease, we will set $\rho = \left(k\epsilon^{-1}\log n\right)^{\delta}$ for some $\delta > 1$.

Pseudocode of our hopset construction algorithm is given in Algorithm 14. Two additional parameters are needed to control the first and last level of the recursion: $\beta = \beta_0$ is the decomposition parameter on the top level, and n_{final} is the base case size at which the recursion stops.

We start with the following simple claim about the β parameters in the recursion.

Claim 5.2.1. *If the top level of the recursion is called with $\beta = \beta_0$ as the input parameter. then the parameter β in i^{th}a level, denoted β_i, is given by $\beta_i = \left(k\epsilon^{-1}\log n\right)^{i}\beta_0$.*

We now describe how hopsets are used to speed up the parallel BFS. We prove the lemma in the generalized weighted setting as it will become useful in Section 5.2.2.

Lemma 5.2.2. *Given a weighted graph $G = (V, E, w)$ with $|V| = n$ and $|E| = m$, let E' be the set of edges added by running* HopSet(V, E, β_0). *Then for any $u, v \in V$, we have with probability at least $1/2$:*

$$dist^{h}_{E\cup E'}(u,v) \leq dist_E(u,v) + O(\epsilon \log_{\rho} n \cdot dist_E(u,v))$$

where $h = n_{final}^{1-1/\delta} n^{1/\delta} \beta_0 dist_E(u,v)$.

Proof. Let p be any shortest path with endpoints u and v, we show how to transform it into a path p' satisfying the above requirements using edges in E'. In each level of the algorithm, the clustering routine breaks p up into smaller pieces by cutting some edges of p. Consider an input subgraph in the recursion that intersects the path p from vertex x to vertex y. The decomposition partitions this intersection into a number of segments, each contained in a cluster. Starting from x, we can identify the first segment that is contained in a large cluster, whose start point is denoted by x', and similarly we can find the last segment contained in a large cluster with its end point denoted by y'. We drop all edges on p between x' and y' and reconnect them using three edges $(x', c(x'))$, $(c(x'), c(y'))$ and $(c(y'), y')$. We will refer to this procedure as short-cutting. We then recursively build the shortcuts on each segment before u' and after v'. Note that these segments are all contained in small clusters, thus they are also recursed on during the hopset construction. We stop at the base case of our hopset algorithm.

We first analyze the number of edges in the final path p', obtained by replacing some portion of p with shortcut edges. The path p' consists of edges cut by the decomposition, shortcut edges that we introduced, and segments that are contained in base case pieces. It suffices to bound the number of cut edges, as the segments in p' separated by the cut edges have size at most the size of the base case. Recall from Corollary 2.2.4 that any edge of length $l(e)$ has probability $\beta l(e)$ of being cut in the clustering. Thus, the expected number of cut edges can be bounded by

$$\sum_{e \in p} \left(\sum_i \beta_i \right) l(e) = \left(\sum_i \beta_i \right) l(p).$$

Since β_is are geometrically increasing, we can use the approximation $\sum_i \beta_i \approx \beta_l$, where $l = \log_\rho n$ is the depth of recursion. Recalling that $\rho = (k\epsilon^{-1} \log n)^\delta$:

$$\begin{aligned} \beta_l d &= \left(k\epsilon^{-1} \log n \right)^{\log_\rho \left(\frac{n}{n_{\mathit{final}}} \right)} \beta_0 d \\ &= \left(\rho^{1/\delta} \right)^{\log_\rho \left(\frac{n}{n_{\mathit{final}}} \right)} \beta_0 d \\ &= \left(\frac{n}{n_{\mathit{final}}} \right)^{1/\delta} \beta_0 d. \end{aligned}$$

As the recursion terminates when clusters have fewer than n_{final} vertices, each path in such a cluster can have at most n_{final} hops. Multiplying in this factor gives $n^{1/\delta} n_{\mathit{final}}^{1-1/\delta} \beta_0 d$.

Next we analyze the distortion introduced by p' compared to the original path p. Shortcutting in level i introduces an additive distortion of at most $4c\beta_i^{-1} \log n$. The expected number of shortcut made in level i, in other words the expected number of cluster in $(i-1)^{\text{th}}$ level intersecting the path p, is bounded by $\beta_{i-1} d$. Thus the amount of additive distortion introduced in level i is at most

$$(\beta_{i-1} d) \cdot \frac{4c \log n}{\beta_i} = O(\epsilon d).$$

This gives an overall additive distortion of $O(\epsilon d \log_\rho n)$.

■

Lemma 5.2.3. *If* HopSet *is run on a graph G with n vertices, it adds at most n star edges and $O\left((n/n_{\mathit{final}}) \log^{2\delta} n \right)$ clique edges to G.*

Proof. As we do not recurse on large clusters, each vertex is part of a large cluster at most once. As a result, we add at most n edges as star edges in Line 17 of HopSet.

To bound the number of clique edges, we claim that the worst case is when we always generate small clusters, except in the level above the base cases, where all the clusters are large.

Suppose an adversary trying to maximize the number of clique edges decides which clusters are large. Since we do not recurse on large clusters, if on any level above the base case we have a large cluster, the adversary can always replace it with a small cluster, losing at most ρ clique edges doing so (since there are at most ρ large clusters), and gain ρ^2 edges in the next level by making the algorithm recurse on that cluster. Since the base case clusters have size at most n_{final}, there are at most n/n_{final} clusters in the level above, where each cluster adding at most ρ^2 edges. Therefore at most $(n/n_{final})\rho^2 = (n/n_{final})(\log n/\epsilon)^{2\delta}$ edges are added in total.

■

Theorem 5.2.4. *Given constants $\delta > 1$ and $\gamma_1 < \gamma_2 < 1$, we can construct a $(\epsilon \log n, h, O(n))$-hopset on a graph with n vertices and m edges in $O(n^{\gamma_2} \log^2 n \log^* n)$ depth and $O(m \log^{1+\delta} n\epsilon^{-\delta})$ work, where $h = n^{1+1/\delta+\gamma_1(1-1/\delta)-\gamma_2}$.*

Proof. The theorem statement can be obtained by setting $\beta_0 = n^{-\gamma_2}$ and $n_{final} = n^{\gamma_1}$. The correctness of the constructed hopset follows directly from Lemma 5.2.2, Lemma 5.2.3, and the fact that any path in an unweighted graph has length at most n. Specifically, for any vertices u and v with $\text{dist}(u, v) = d$, the expected hop-count is:

$$\begin{aligned} n^{1/\delta} n_{final}^{1-1/\delta} \beta_0 d &\leq n^{1/\delta} n^{\gamma_1(1-1/\delta_2)} n^{-\gamma_2} n \\ &= n^{1+1/\delta+\gamma_1(1-1/\delta)-\gamma_2} \end{aligned}$$

and the expected distortion is $O(\epsilon \log n \cdot d)$. By Markov's inequality, the probability of both of these exceeding four times their expected value is at most $1/2$, and the result can be obtained by adjusting the constants.

So we focus on bounding the depth and work. As the size of each cluster decreases by a factor of ρ from one level to the next, the number of recursion levels is bounded by $\log_\rho(n/n_{final})$. As n/n_{final} is polynomial in n with our choice of parameters, we will treat this term as $\log n$.

The algorithm starts by calling HopSet$(V, E, n^{-\gamma_2})$. Since the recursive calls are done in parallel, it suffices to bound the time spent in a single call on each level. Theorem 2.3.2 gives that the clustering takes $O(\beta^{-1} \log n \log^* n)$ depth and linear work. Since the value of β only increases in subsequent levels, all decompositions in each level of the recursion can be computed in $O(n^{\gamma_2} \log n \log^* n)$ depth and $O(m)$ work. This gives a total of $O(n^{\gamma_2} \log^2 n \log^* n)$ depth and $O(m \log n)$ work from Line 4. In addition, Line 17 can be easily incorporated into the decomposition routine at no extra cost.

To compute the all-pair shortest distances between the centers of the large clusters (Line 22), we perform the parallel BFS by [UY91] from each of the centers. By Theorem 2.2.5, the diameter of the input graphs to recursive calls after the top level is bounded by $O(n^{\gamma_2} \log n)$. Therefore the parallel BFS only need to be ran for $O(n^{\gamma_2} \log n)$ levels. This gives a total depth

of $O(n^{\gamma_2} \log n \log^* n)$ and work of $O(\rho m)$ per level. Summing over $O(\log n)$ levels of recursion gives $O(n^{\gamma_2} \log^2 n \log^* n)$ depth and $O(\rho m \log n) = O(\epsilon^{-\delta} m \log^{1+\delta} n)$ work.

■

The unweighted version of Theorem 5.1.2 then follows from Theorem 5.2.4 by setting $\delta = 1 + \alpha$, and solving $h = n^{\gamma_2}$ to balance the depth for hopset construction and the depth for finding approximate distances using hopsets [KS97]. For a concrete example of setting these parameters, $\delta = 1.1$, $\epsilon = \frac{\epsilon'}{\log n}$, $\gamma_2 = 0.96$, and setting γ_1 to some small constant leads to the following bound.

Corollary 5.2.5. *For any constant $\epsilon' > 0$, there exists an algorithm for finding $(1+\epsilon')$-approximation to unweighted $s - t$ shortest path that runs in $O(n^{0.96} \log^2 n \log^* n)$ depth and $O(m \log^{3.2} n)$ work.*

5.2.2 Hopsets in Weighted Graphs

In this section we show how to construct hopsets in weighted graphs. We will assume that the ratio between the longest and the shortest edge lengths is $O(n^3)$. This is due to a reduction similar to the one by Klein and Subramanian [KS97] where they partitioned the edges into buckets with lengths between powers of 2, and show that only considering edges from $O(\log n)$ consecutive buckets suffices for approximate shortest path computation. This scheme can be modified by choosing buckets with powers of n, and then considering a constant number of consecutive buckets suffices for good approximations. This result is summarized in the following lemma and a proof is provided in Appendix 5.3 for completeness.

Lemma 5.2.6. *Given a weighted graph $G = (V, E, w)$, we can efficiently construct a collection of graphs with $O(|V|)$ vertices and $O(|E|)$ edges in total, such that the edge lengths in any one of these graphs are within $O(n^3)$ of each other. Furthermore, given a shortest path query, we can map it to a query on one of the graphs whose answer is a $(1-\epsilon)$-approximation for the original query.*

A simple adaptation of parallel BFS to weighted graphs can lead to depth linear in path lengths, which can potentially be big even though the number of edge hops is small. To alleviate this we borrow a rounding technique from [KS97]. The main idea is to round up small edge lengths and pay a small amount of distortion, so that the search advances much faster.

Suppose we are interested in a path p with at most k edges whose length is between d and cd. We can perturb the length of each edge additively by $\frac{\zeta d}{k}$ without distorting the final length by more than ζd. This value serves as the "granularity" of our rounding, which we denote using $\hat{w}$:

$$\hat{w} = \frac{\zeta d}{k}$$

for some $0 < \zeta < 1$ and round the edge lengths $l(e)$ to $\tilde{l}(e)$

$$\tilde{l}(e) = \left\lceil \frac{l(e)}{\hat{w}} \right\rceil.$$

Notice that this rounds edge lengths to multiples of $\hat{w}$. The properties we need from this rounding scheme is summarized in the following lemma.

Lemma 5.2.7 (Klein and Subramanian [KS97]). *Given a weighted graph and a number d. Under the above rounding scheme, any shortest path p with size at most k and length $d \leq l(p) \leq cd$ for some c in the original graph now has length $\tilde{l}(p) \leq \lceil ck/\zeta \rceil$ and $\hat{w} \cdot \tilde{l}(p) \leq (1+\zeta)l(p)$.*

Thus we only need to run weighted parallel BFS for $O(ck\zeta^{-1})$ levels to recover p, giving a depth of $O(ck\zeta^{-1} \log n)$. Therefore, if we set $c = n^{\eta}$ for some $\eta < 1$, and since the edge lengths are within $O(n^3)$ of each other, we can just try building hopsets using $O(3/\eta)$ estimates, incurring a factor of $O(3/\eta)$ in the work. As one of the values tried satisfies $d \leq l(p) \leq cd$, Lemma 5.2.7 gives that if ζ is set to $\epsilon/2$, an $(1+\epsilon/2)$-approximation of the shortest path in the rounded graph is in turn an $(1+\epsilon)$-approximation to the shortest path in the original graph. Therefore, from this point on we will focus on finding an $(1+\epsilon)$-approximation of the shortest path in the rounded graph with lengths $\tilde{w}(e)$. In particular, we have that all edge lengths are positive integers, and the shortest path between s and t has length $O(n^{1+\eta}/\zeta) = O(n^{1+\eta}/\epsilon)$.

Theorem 5.2.8. *For any constants $\delta > 1$ and $\gamma_1 < \gamma_2 < 1$, we can construct a $(\epsilon \log n, h, O(n))$-hopset on a graph with n vertices and m edges in expected $O((n/\epsilon)^{\gamma_2} \log^2 n \log^* n)$ depth and $O(m \log^{1+\delta} n\epsilon^{-\delta})$ work, where $h = n^{1+1/\delta+\eta+\gamma_1(1-1/\delta)-\gamma_2}/\epsilon^{1-\gamma_2}$.*

Proof. Since the edge lengths are within a polynomial of each other, we can build $O(1/\eta)$ hopsets in parallel for all values of d being powers of n^{η}. For any pair of vertices s and t, one of the value tried will satisfy $d \leq \text{dist}(s,t) \leq n^{\eta}d$. Given such an estimate, we first perform the rounding described above, then we run Algorithm 14 with $\beta = (n/\epsilon)^{-\gamma_2}$ and $n_{final} = n^{\gamma_1}$. The exponential start time clustering in Line 4 takes place in the weighted setting, and Line 22 becomes a weighted parallel BFS. The correctness of the hopset constructed follows from Lemma 5.2.2, Lemma 5.2.3, and the fact that $\text{dist}(s,t) = O(n^{1+\eta}/\epsilon)$ by the rounding. Specifically, the expected hop count is

$$\begin{aligned} n^{1/\delta} n_{final}^{1-1/\delta} \beta d &\leq n^{1/\delta} n^{\gamma_1(1-1/\delta)} \left(\frac{n}{\epsilon}\right)^{-\gamma_2} \frac{n^{1+\eta}}{\epsilon} \\ &= n^{1+1/\delta+\eta+\gamma_1(1-1/\delta)-\gamma_2}/\epsilon^{1-\gamma_2} \end{aligned}$$

and the expected distortion is $O(\epsilon d)$. By Markov's inequality, the probability of both of these exceeding four times their expected values is at most $1/2$.

The number of recursion levels is still bounded by $\log_{\rho} n$. Since the βs only increase, according to Theorem 2.3.2 we spend $O((n/\epsilon)^{\gamma_2} \log n \log^* n)$ depth in each level of the recursion

and $O((n/\epsilon)^{\gamma_2} \log^2 n \log^* n)$ overall in Line 4. Since our decomposition is laminar, we spend $O(m)$ work in each level and $O(m \log n)$ overall in Line 4. Again, Line 17 can be incorporated into the decomposition with no extra cost.

Since the diameter of the pieces below the top level is bounded by $\beta^{-1} \log n = (n/\epsilon)^{\gamma_2} \log n$ and the minimum edge length is one, Line 22 can be implemented by weighted parallel BFS in depth $O((n/\epsilon)^{\gamma_2} \log n \log^* n)$ in one level and $O((n/\epsilon)^{\gamma_2} \log^2 n \log^* n)$ in total. The work done by the weighted parallel BFS is $O(\rho m)$ per level and $O(\rho m \log n) = O(m \log^{1+\delta} n \epsilon^{-\delta})$ in total.

■

Theorem 5.1.2 then follows from Theorem 5.2.8 by adjusting the various parameters. Again, to give a concrete example, we can set $\delta = 1.1$, $\epsilon = \epsilon'/(\log n)$, $\gamma_2 = 0.96$, and set γ_1 and ζ to some small constants to obtain the following corollary

Corollary 5.2.9. *For any constant error factor ϵ', there exists an algorithm for finding $(1+\epsilon')$-approximation to weighted s-t shortest path that runs in $O(n^{0.96} \log^2 n \log^* n)$ depth and $O(m \log^{3.2} n)$ work in a graph with polynomial edge length ratio.*

Notice that with our current scheme it is not possible to push the depth under $\tilde{O}(\sqrt{n})$ as the hop count becomes the bottle neck. A modification that allows us to obtain a depth of $\tilde{O}(n^{\alpha})$ for arbitrary $\alpha > 0$ at the expense of incurring more work can be found in Appendix 5.4.

5.3 Preprocessing of Weighted Graphs

Here we describe the reduction needed for the assumption of edge lengths being polynomially bounded in Section 5.2.2. We will present a scheme for transforming a graph into a collection of graphs where the ratio between the maximum and minimum edge lengths is at most $O(n^3)$ in each graph. The total size of this collection is on the order of the original graph size, and given any query, we can map it to a query on one of the graphs in this collection efficiently. The technique presented is similar to the scheme used by Klein and Sairam [KS97]. They partition edges into categories with lengths between consecutive powers of 2, and show that only considering edges from $O(\log n)$ consecutive categories suffice for approximate shortest path computation. We modify this scheme slightly by choosing categories by powers of n, and show that picking a constant number of consecutive categories suffice.

We will divide the edges into successive categories according to their lengths, so that edge lengths from non-consecutive categories differ significantly. If the shortest path needs to use an edge from a category of very long length, any edges in shorter length categories can be discarded with minor distortion. Thus setting lengths in these shorter categories to 0 does not change the answer too much. As the graph is undirected, this is equivalent to constructing

the quotient graph formed by contracting these edges. We then show that the total size of these quotient graphs over all categories is small. This allows us to precompute all of them beforehand, and use hopsets for one of them to answer each query. To simplify notations when working with these quotient graphs, we use G/E' to denote the quotient graph formed by contracting a subset of edges $E' \subseteq E$.

Given a weighted graph $G = (V, E, w)$, we may assume that the minimum edge length is 1 by simplying renormalizing. Then we group the edges into categories as follows:

$$E_i = \left\{ e \in E \mid (n/\epsilon)^i \leq l(e) < (n/\epsilon)^{i+1} \right\}.$$

As the contractions are done to all edges belonging to some lower category, they correspond to prefixes in this list of categories. We will denote these using $P_j = \bigcup_{i=0}^{j} E_i$. Also, let $q(1), \cdots, q(k)$ be the indices of the non-empty categories in G. Contracting $E_1, E_2 \ldots$ in order leads to a laminar decomposition of the graph, which we formalize as a hierarchical length decomposition:

Definition 5.3.1. *A hierarchical length decomposition is defined inductively as follows.*

- *The vertices form the leaves. For convenience we say that the leaves form the 0th level and define $E_{q(0)} = \varnothing$.*
- *Given the jth level whose nodes represent connected components of $G[E_{q(j)}]$, we form the $(j+1)^{th}$ level by adding a node for each connected components of $G[E_{q(j+1)}]$, and make it the parent of the components in $G[E_{q(j)}]$ it contains.*

Lemma 5.3.2. *A hierarchical weight decomposition can be computed in $O(\log^3 n)$ depth and $O(m \log n)$ work.*

Proof. We first compute the non-empty categories $E_{q(1)}, \cdots, E_{q(k)}$ where $k \leq m$. Then we perform divide and conquer on the number of weight categories. Let E_j be the median weight class, the connected components of $G[E_j]$ can be computed using the graph connectivity algorithm by Gazit [Gaz93] in $O(\log n)$ depth and $|E_j|$ work. We then recurse on each connected components and also on the quotient graph where all the components of $G[E_j]$ are collapsed to a point.

∎

This then allows us to prove Lemma 5.2.6 at the start of Section 5.2.2 about only working with graphs with polynomially bounded edge weights.

of Lemma. We first construct the decomposition tree from Definition 5.3.1. Once we have the tree, given a query on the distance between s and t, we can find their least common ancestor (LCA) in the tree using parallel tree contraction. Let j be the level the LCA of s and t is in,

then we claim that we only need to consider edges in $E_{q(j-1)} \cup E_{q(j)} \cup E_{q(j+1)}$. Since the LCA is in j^{th} level, the $s-t$ shortest path uses at least one edge, say e_j, from $E_{q(j)}$. By definition, for any edge $e_{j-2} \in P_{q(j-2)}$, we have $(n/\epsilon)l(e_{j-2}) \leq l(e_j)$. Since the $s-t$ path can have at most $n-1$ edges, setting lengths of edges in $E_{q(j-2)}$ to 0 incurs a multiplicative distortion of at most ϵ. Moreover, edges in level $j+2$ and above have weights at least $(n/\epsilon)l(e')$, and since s and t is in one connected components of $G[E_{q(j)}]$, no edge in level $j+2$ and above can be part of the $s-t$ shortest path.

Consider the induced subgraph $G[P_{q(j+1)}]$ and its quotient graph where all edges in $P_{q(j-1)}$ are collapsed to points: $G[P_{q(j+1)}]/P_{q(j-1)}$. Let s' be the component in $G[E_{q(j-1)}]$ containing s and let t' be the component that contains t. By the above argument, the shortest path between s' and t' in $G[P_{q(j+1)}]/P_{q(j-1)}$ is an $(1-\epsilon)$-approximation for the $s-t$ shortest path in G. Lemma 5.3.2 allows us to build the graphs $G[P_{q(j+1)}]/P_{q(j-1)}$ for all j as part of the decomposition tree construction without changing the total cost of constructing the hopsets. Each edge of G appears at most three times in these quotient graphs, however the number of vertices is equal to the size of the decomposition tree. We trim down this number by observing that any chain in the tree of length more than three can be shortened to length three by throwing out the middle parts as they will never be used for any query. This gives an overall bound of $O(|V|)$.

■

5.4 Obtaining Lower Depth

We now show that the depth of our algorithms can be reduced arbitrarily to n^{α} for any $\alpha > 0$ in ways similar to the Limited-Hopset algorithm by Cohen [Coh00]. So far, we have been trying to approximate paths of potentially n hops with paths of much fewer hops. Consider the bound from Theorem 5.2.4, which gives a hop count of $n^{1+1/\delta+\gamma_1-\gamma_2}$ for $\delta > 1$ and $\gamma_1 < \gamma_2 < 1$. The the first factor of n is a result of handling path containing up to n hops directly. We now show a more gradual scheme that gradually reduces the length of these paths. Instead of reducing the hop-count of paths containing up to n edges, we approximate $n^{2\eta}$-hop paths with ones containing n^{η} hops for some small η. This routine can be applied to a longer path with k hops, by breaking it into $kn^{-2\eta}$ ones of $n^{2\eta}$ hops each and apply the guarantee separately. If the guarantee holds deterministically, we get an approximation with $kn^{-\eta}$ hops. Repeating this for $1/\eta$ steps would then lead to a low depth algorithm. However, the probabilistic guarantees of our algorithms make it necessary to argue about the various piece simultaneously. This avoids having probabilistic bounds on each piece separately, but rather one per weight class.

Lemma 5.4.1. *Given a graph $G = (V, E, w)$, let $p_1 \dots p_t$ be a collection of disjoint paths hidden from the program such that each p_i has at most $k = n^{2\eta}$ hops and weight between d and dn^{η}. For any failure probability $p_{failure}$, we can construct in $O(n^{\eta}/\epsilon)$ depth and $O(m \log^{2+\frac{2}{\eta}} n/\epsilon)$ work a set of at most*

$O(n^{1-\frac{\eta}{2}})$ *edges* E' *such that with probability at least* $1 - p_{failure}$ *there exist paths* $p'_1 \ldots p'_t$ *such that:*

1. p'_i *starts and ends at the same vertices as* p_i.
2. *The total number of hops in* $p'_1 \ldots p'_t$ *can be bounded by* tn^η.
3. $\sum_{i=1}^{t} l(p'_i) \leq (1+\epsilon) \sum_{i=1}^{t} l(p_i)$.

Proof. We use the rounding scheme and construction for weighted paths from Section 5.2.2. We first round the edge weights with $\hat{w} = \epsilon d n^{-2\eta}$. As the paths have at most $k = n^{2\eta}$ edges, the guarantees of Lemma 5.2.7 gives that the lengths of paths are distorted by a factor of $(1+\epsilon)$. Furthermore, this rounding leaves us with integer edge weights such that the total length of each path is at most $d = n^{3\eta}/\epsilon$.

We can then call Algorithm 14 on the rounded graph with the following parameters:

$$\delta = \frac{2}{\eta},$$
$$\beta_0 = \left(\frac{n^{3\eta}}{\epsilon}\right)^{-1} = \frac{1}{d},$$
$$n_{final} = n^{\frac{\eta}{2}},$$
$$\epsilon' = \frac{\epsilon}{\log n}.$$

By an argument similar to the proof of Theorem 5.2.8 and Lemma 5.2.2, this takes $O((n^{2\eta}/\epsilon)\log n \log_K n\epsilon)$ depth and $O(m \log^{1+\delta} n/\epsilon') = O(m \log^{2+\frac{2}{\eta}} n/\epsilon)$ work. Furthermore, for each p_i, the expected number of pieces that it is partitioned into is:

$$n^{\frac{1}{\delta}} \beta_0 d n_{final} = n^{\frac{\eta}{2}} n^{\frac{\eta}{2}} = n^\eta$$

and if we take all shortcuts through centers of big clusters, the expected distortion is:

$$O(\log_{\rho_n} n\epsilon' d) = \epsilon d.$$

Applying linearity of expectation over all t paths gives that the expected total number of hops is tn^η, and the expected additive distortion is ϵdt. As both of these values are non-negative, Markov's inequality the probability of any of these exceeding $\frac{2}{p_{failure}}$ of their expected value is at most $p_{failure}$. Therefore, adjusting the constants and ϵ accordingly gives the guarantee. Finally, the number of edges in the hopset can be bounded by $n^{1-\eta} \log^{\frac{4}{\eta}} n \leq n^{1-\frac{\eta}{2}}$.

■

Then it suffices to run this routine for all values of d equaling to powers of n^η. The fact that edge weights are polynomially bounded means that this only leads to a constant factor

overhead in work. Running this routine another n^η times gives the hopset paths with arbitrary number of hops.

Theorem 5.4.2. *Given a graph $G = (V, E, w)$ with polynomially bounded edge weights and any constant $\alpha > 0$, we can construct a $(\epsilon, n^\alpha, O(n))$-hopset for G in $O(n^\alpha \epsilon^{-1})$ depth and $O(m \log^{O(\frac{1}{\alpha})} n\epsilon^{-1})$ work*

Proof. Let $\eta = \alpha/2$. We will repeat the following $\frac{1}{\eta}$ times: run the algorithm given in Lemma 5.4.1 repeatedly for all values of d being powers of n^η, and add the edges of the hopset to the current graph. As the edge weights are polynomially bounded, there are $O(\frac{1}{\eta^2})$ invocations, and we can choose the constants to that they can all succeed with probability at least $1/2$. In this case, we will prove the guarantees of the final set of edges by induction on the number of iterations.

Consider a path p with k hops. If $k \leq n^{2\eta}$, then the path itself serves as a k-hop equivalent. Otherwise, partition the path into pieces with $n^{2\eta}$ hops, with the exception of possibly the last $n^{2\eta}$ edges. Consider these sub-paths classified by their weights. The guarantees of Lemma 5.4.1 gives that each weight class can be approximated with a set of paths containing $n^{-\eta}$ as many edges. Putting these shortcuts together gives that there is a path p' with $kn^{-\eta}$ hops such that $l(p') \leq (1+\epsilon)l(p)$. Since p' has fewer edges, applying the inductive hypothesis gives that p' has an equivalent in the final graph with $n^{2\eta}$ hops that incurs a distortion of $(1 + (\log_{n^\eta} k - 1)\epsilon)$. Multiplying together these two distortion factors gives that this path also approximates p with distortion $(1 + \log_{n^\eta} k\epsilon)$. As $k \leq n$ and η is a constant, replacing ϵ with $\eta\epsilon$ gives the result.

■

Algorithm 14 UNWEIGHTED-HOPSET(G, β)

1: **if then** $|V| \leq n_{final}$
2: **return** $\varnothing$
3: **end if**
4: $\mathcal{X} \leftarrow$ ESTCLUSTER(G, β)
5: **if** this is the top level call **then**
6: **for** each $X \in \mathcal{X}$, in parallel **do**
7: $H_X \leftarrow$ UNWEIGHTED-HOPSET$(X, (k\epsilon^{-1} \log n)\beta)$
8: **end for**
9: **return** $\bigcup_{X \in \mathcal{X}} H_X$
10: **else**
11: $\mathcal{X}_b \leftarrow \{X \in \mathcal{X} : |X| \geq |V|/\rho\}$ (the set of large clusters)
12: $\mathcal{X}_s \leftarrow \{X \in \mathcal{X} : |X| < |V|/\rho\}$ (the set of small clusters)
13: $H \leftarrow \varnothing$
14: **for** each large cluster $X \in \mathcal{X}_l$ **do**
15: let c be the center of X
16: **for** vertex $v \in X$ **do**
17: add a star edge between v and c with length $\text{dist}(v, c)$ to H
18: **end for**
19: **end for**
20: **for** all pairs of large clusters $X_1, X_2 \in \mathcal{X}$ **do**
21: let c_1 and c_2 be the centers of X_1 and X_2 respectively
22: add a clique edge between c_1 and c_2 with length $\text{dist}(c_1, c_2)$ to H
23: **end for**
24: **for** each small cluster $X \in \mathcal{X}_s$, in parallel **do**
25: $H_X \leftarrow$ UNWEIGHTED-HOPSET$(X, (k\epsilon^{-1} \log n)\beta)$
26: **end forreturn** $H \cup (\bigcup_{X \in \mathcal{X}_s} H_X)$
27: **end if**

Chapter 6

Conclusions and Open Problems

In this thesis we presented efficient combinatorial algorithms that study the spectral properties of graph Laplacians. Specifically we gave a simple parallel algorithm based on exponential start time clustering for computing low diameter decompositions. We then applied this algorithm to obtain improved constructions of graph spanners, spectral sparsifiers, and low stretch tree embeddings suitable for fast graph Laplacian solvers. Given the ubiquitous applications of graph Laplacians as well as the increasing number of recent theoretical work in the field of spectral graph theory, the practicality of these algorithms becomes an important and intriguing question.

Our exponential start time algorithm for low diameter graph decomposition already saw a few practical successes due to its simplicity and parallel nature. Shun *et al.* [SDB14] gave an efficient parallel implementation of this algorithm and applied it to the problem of parallel graph connectivity. The closely related algorithms for graph spanners and sparsifiers have also been applied to Laplacian smoothing by Sadhanala *et al.* [SWT16]. As for the other major application of these algorithms, solving linear systems in graph Laplacians and SDD matrices, is the topic of much ongoing and future work.

The first nearly linear time graph Laplacian solver studied in a practical setting is the flow based solver of Kelner *et al.* [KOSZ13]. Recall from Section 1.1.2 that solving the linear system in graph Laplacian $\mathbf{L}x = b$ can be viewed as finding a voltage setting (the unknown vector x), such that the induced electric flow satisfies the demand on the net incoming/outgoing flow at each vertex specified by b. It turns out that this induced flow has the minimum electric energy among all flows that satisfy the demand, and the Kelner *et al.* solver tries to directly find such a flow. They start out by finding the (unique) flow that meets the demand on a low stretch spanning tree, and iteratively improve its energy while staying as a feasible flow, by updating the flow along fundamental cycles of the tree. Hoske *et al.* [HLMW15] conducted an experimental work on this flow based solver and observed that the tree data structure is

the major bottleneck of this algorithm. Subsequently Deweese *et al.* [DGM+16] studied this approach on the class of graphs where the low stretch spanning trees are a (Hamiltonian) path. This assumption simplifies the data structure problem, and combined with a more efficient tree data structure, we observed that the performance is competitive with some standard numeric routines.

The natural next step is then to investigate the practicality of the recursive combinatorial preconditioning framework based on low stretch trees. Compared to other existing combinatorial preconditioning packages such as combinatorial multigrid [KMT11], this method have a few potential drawbacks. First it relies on the ability to find high quality low stretch trees, which is still a quite difficult problem in terms of having practical implementations. The other potential drawback is the higher number of recursive calls as well as the more expensive iterative method used, which can affect the running time by constant factors in practice. Thus it would be beneficial to start by examining artificial graphs with low stretch trees already built-in, and focus our attention on understanding with what types of graphs does being provably fast gives this framework an edge over existing techniques. Alternatively, one could also explore possible practical compromises in order to obtain a fast solver, while still drawing ideas developed from the literature.

Bibliography

[ABCP98] Baruch Awerbuch, Bonnie Berger, Lenore Cowen, and David Peleg. Near-linear time construction of sparse neighborhood covers. *SIAM Journal on Computing*, 28(1):263–277, 1998.

[ABN08] Ittai Abraham, Yair Bartal, and Ofer Neiman. Nearly tight low stretch spanning trees. In *Proceedings of the 2008 49th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '08, pages 781–790, Washington, DC, USA, 2008. IEEE Computer Society.

[ADD+93] Ingo Althöfer, Gautam Das, David Dobkin, Deborah Joseph, and José Soares. On sparse spanners of weighted graphs. *Discrete Comput. Geom.*, 9(1):81–100, January 1993.

[AKPW95] Noga Alon, Richard M. Karp, David Peleg, and Douglas West. A graph-theoretic game and its application to the k-server problem. *SIAM J. Comput.*, 24(1):78–100, February 1995.

[AN12] Ittai Abraham and Ofer Neiman. Using petal-decompositions to build a low stretch spanning tree. In *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing*, STOC '12, pages 395–406, New York, NY, USA, 2012. ACM.

[Awe85] Baruch Awerbuch. Complexity of network synchronization. *J. ACM*, 32(4):804–823, October 1985.

[Bar96] Yair Bartal. Probabilistic approximation of metric spaces and its algorithmic applications. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, FOCS '96, pages 184–, Washington, DC, USA, 1996. IEEE Computer Society.

[Bar98] Yair Bartal. On approximating arbitrary metrices by tree metrics. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pages 161–168, New York, NY, USA, 1998. ACM.

[BGK+14] Guy E. Blelloch, Anupam Gupta, Ioannis Koutis, Gary L. Miller, Richard Peng, and Kanat Tangwongsan. Nearly-linear work parallel SDD solvers, low-diameter decomposition, and low-stretch subgraphs. *Theor. Comp. Sys.*, 55(3):521–554, October 2014.

[BH01] Eric Boman and Bruce Hendrickson. On spanning tree preco nditioners. Manuscript, Sandia National Laboratories, 2001.

[BS07] Surender Baswana and Sandeep Sen. A simple and linear time randomized algorithm for computing sparse spanners in weighted graphs. *Random Struct. Algorithms*, 30(4):532–563, July 2007.

[BSS12] Joshua Baston, Daniel A. Spielman, and Nikhil Srivastava. Twice-ramanujan sparsifiers. *SIAM Journal on Computing*, 41(6):1704–1721, 2012.

[CCG+98] Moses Charikar, Chandra Chekuri, Ashish Goel, Sudipto Guha, and Serge Plotkin. Approximating a finite metric by a small number of tree metrics. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, FOCS '98, pages 379–, Washington, DC, USA, 1998. IEEE Computer Society.

[CFM+14] Michael B. Cohen, Brittany Terese Fasy, Gary L. Miller, Amir Nayyeri, Richard Peng, and Noel Walkington. Solving 1-laplacians in nearly linear time: Collapsing and expanding a topological ball. In *Proceedings of the Twenty-fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '14, pages 204–216, Philadelphia, PA, USA, 2014. Society for Industrial and Applied Mathematics.

[CKM+11] Paul Christiano, Jonathan A. Kelner, Aleksander Madry, Daniel A. Spielman, and Shang-Hua Teng. Electrical flows, laplacian systems, and faster approximation of maximum flow in undirected graphs. In *Proceedings of the 43rd Annual ACM Symposium on Theory of Computing*, STOC '11, pages 273–282, New York, NY, USA, 2011. ACM.

[CKM+14] Michael B. Cohen, Rasmus Kyng, Gary L. Miller, Jakub W. Pachocki, Richard Peng, Anup B. Rao, and Shen Chen Xu. Solving SDD linear systems in nearly $m \log^{1/2} n$ time. In *Proceedings of the Forty-sixth Annual ACM Symposium on Theory of Computing*, STOC '14, New York, NY, USA, 2014. ACM.

[CKR05] Gruia Calinescu, Howard Karloff, and Yuval Rabani. Approximation algorithms for the 0-extension problem. *SIAM J. Comput.*, 34(2):358–372, February 2005.

[CMP+14] Michael B. Cohen, Gary L. Miller, Jakub W. Pachocki, Richard Peng, and Shen Chen Xu. Stretching stretch. *CoRR*, abs/1401.2454, 2014.

[CMSV16] Michael B. Cohen, Aleksander Madry, Piotr Sankowski, and Adrian Vladu. Negative-weight shortest paths and unit capacity minimum cost flow in $\tilde{O}(m^{10/7} \log W)$ time. *CoRR*, abs/1605.01717, 2016.

[Coh98] Edith Cohen. Fast algorithms for constructing t-spanners and paths with stretch t. *SIAM Journal on Computing*, 28(1):210–236, 1998.

[Coh00] Edith Cohen. Polylog-time and near-linear work approximation scheme for undirected shortest paths. *J. ACM*, 47(1):132–166, January 2000.

[DGM+16] Kevin Deweese, John R. Gilbert, Gary L. Miller, Richard Peng, Hao Ran Xu, and Shen Chen Xu. *An Empirical Study of Cycle Toggling Based Laplacian Solvers*, pages 33–41. 2016.

[Dia69] Robert B. Dial. Algorithm 360: Shortest-path forest with topological ordering [h]. *Commun. ACM*, 12(11):632–633, November 1969.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1(1):269–271, December 1959.

[DS84] Peter G. Doyle and J. Laurie Snell. *Random Walks and Electric Networks*, volume 22 of *Carus Mathematical Monographs*. Mathematical Association of America, 1 edition, 1984.

[DS08] Samuel I. Daitch and Daniel A. Spielman. Faster approximate lossy generalized flow via interior point algorithms. In *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing*, STOC '08, pages 451–460, New York, NY, USA, 2008. ACM.

[EEST08] Michael Elkin, Yuval Emek, Daniel A. Spielman, and Shang-Hua Teng. Lower-stretch spanning trees. *SIAM J. Comput.*, 38(2):608–628, May 2008.

[EN16] Michael Elkin and Ofer Neiman. Distributed strong diameter network decomposition: Extended abstract. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC '16, pages 211–216, New York, NY, USA, 2016. ACM.

[Erd64] Paul Erdős. Extremal problems in graph theory. In *In Theory of Graphs and its Applications*, Proc. Sympos. Smolenice, 1963, pages 29–36, Prague, 1964. Czechoslovak Acad. Sci.

[FHRT03] Jittat Fakcharoenphol, Chris Harrelson, Satish Rao, and Kunal Talwar. An improved approximation algorithm for the 0-extension problem. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 257–265, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[Fos49] Ronald M Foster. The average impedance of an electrical network. *Contributions to Applied Mechanics (Reissner Anniversary Volume)*, pages 333–340, 1949.

[FRT04] Jittat Fakcharoenphol, Satish Rao, and Kunal Talwar. A tight bound on approximating arbitrary metrics by tree metrics. *J. Comput. Syst. Sci.*, 69(3):485–497, November 2004.

[FT87] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, July 1987.

[Gaz93] Hillel Gazit. Randomized parallel connectivity. In John Reif, editor, *Synthesis of Parallel Algorithms*, chapter 3, pages 115–194. Morgan Kaufmann, 1993.

[GMV91] Joseph Gil, Yossi Matias, and Uzi Vishkin. Towards a theory of nearly constant time parallel algorithms. In *FOCS*, pages 698–710. IEEE Computer Society, 1991.

[Gre96] Keith D. Gremban. *Combinatorial Preconditioners for Sparse, Symmetric, Diagonally Dominant Linear Systems*. PhD thesis, Carnegie Mellon University, 1996.

[GT83] Harold N. Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 246–251, New York, NY, USA, 1983. ACM.

[Han04] Yijie Han. Deterministic sorting in $o(n \log\log n)$ time and linear space. *J. Algorithms*, 50(1):96–105, January 2004.

[HLMW15] Daniel Hoske, Dimitar Lukarski, Henning Meyerhenke, and Michael Wegner. Is nearly-linear the same in theory and practice? a case study with a combinatorial laplacian solver. In *Proceedings of the 14th International Symposium on Experimental Algorithms - Volume 9125*, pages 205–218, New York, NY, USA, 2015. Springer-Verlag New York, Inc.

[KLOS14] J. Kelner, Y. Lee, L. Orecchia, and A. Sidford. An almost-linear-time algorithm for approximate max flow in undirected graphs, and its multicommodity generalizations. In *Proceedings of the 25th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 217–226, 2014.

[KLP15] Ioannis Koutis, Alex Levin, and Richard Peng. Faster spectral sparsification and numerical algorithms for SDD matrices. *ACM Trans. Algorithms*, 12(2):17:1–17:16, December 2015.

[KMP11] Ioannis Koutis, Gary L. Miller, and Richard Peng. A nearly-$m \log n$ time solver for SDD linear systems. In *Proceedings of the 2011 IEEE 52Nd Annual Symposium on Foundations of Computer Science*, FOCS '11, pages 590–598, Washington, DC, USA, 2011. IEEE Computer Society.

[KMP14] Ioannis Koutis, Gary L. Miller, and Richard Peng. Approaching optimality for solving SDD linear systems. *SIAM Journal on Computing*, 43(1):337–354, 2014.

[KMT11] Ioannis Koutis, Gary L. Miller, and David Tolliver. Combinatorial preconditioners and multilevel solvers for problems in computer vision and image processing. *Comput. Vis. Image Underst.*, 115(12):1638–1646, December 2011.

[KOSZ13] Jonathan A. Kelner, Lorenzo Orecchia, Aaron Sidford, and Zeyuan Allen Zhu. A simple, combinatorial algorithm for solving SDD systems in nearly-linear time. In *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing*, STOC '13, pages 911–920, New York, NY, USA, 2013. ACM.

[Kou14] Ioannis Koutis. Simple parallel and distributed algorithms for spectral graph sparsification. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, pages 61–66, New York, NY, USA, 2014. ACM.

[KP12] Michael Kapralov and Rina Panigrahy. Spectral sparsification via random spanners. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pages 393–398, New York, NY, USA, 2012. ACM.

[KS97] Philip N. Klein and Sairam Subramanian. A randomized parallel algorithm for single-source shortest paths. *J. Algorithms*, 25(2):205–220, November 1997.

[KX16] Ioannis Koutis and Shen Chen Xu. Simple parallel and distributed algorithms for spectral graph sparsification. *ACM Trans. Parallel Comput.*, 3(2):14:1–14:14, August 2016.

[Lei92] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Array, Trees, Hypercubes*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[LMR94] Frank Thomson Leighton, Bruce M. Maggs, and Satish B. Rao. Packet routing and job-shop scheduling in $o(\text{congestion} + \text{dilation})$ steps. *Combinatorica*, 14(2):167–186, 1994.

[LRS13] Yin Tat Lee, Satish Rao, and Nikhil Srivastava. A new approach to computing maximum flows using electrical flows. In *Proceedings of the 45th Annual ACM Symposium on Symposium on Theory of Computing*, STOC '13, pages 755–764, New York, NY, USA, 2013. ACM.

[LS13] Yin Tat Lee and Aaron Sidford. Efficient accelerated coordinate descent methods and faster algorithms for solving linear systems. In *Proceedings of the 2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, FOCS '13, pages 147–156, Washington, DC, USA, 2013. IEEE Computer Society.

[LS14] Yin Tat Lee and Aaron Sidford. Path finding methods for linear programming: Solving linear programs in $\tilde{O}(\sqrt{rank})$ iterations and faster algorithms for maximum flow. In *Proceedings of the 2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, FOCS '14, pages 424–433, Washington, DC, USA, 2014. IEEE Computer Society.

[LS17] Yin Tat Lee and He Sun. An sdp-based algorithm for linear-sized spectral sparsification. *CoRR*, abs/1702.08415, 2017.

[Mad10] Aleksander Madry. Fast approximation algorithms for cut-based problems in undirected graphs. In *Proceedings of the 2010 IEEE 51st Annual Symposium on Foundations of Computer Science*, FOCS '10, pages 245–254, Washington, DC, USA, 2010. IEEE Computer Society.

[Mad13] Aleksander Madry. Navigating central path with electrical flows: From flows to matchings, and back. In *Foundations of Computer Science (FOCS), 2013 IEEE 54th Annual Symposium on*, pages 253–262, October 2013.

[MPVX15] Gary L. Miller, Richard Peng, Adrian Vladu, and Shen Chen Xu. Improved parallel algorithms for spanners and hopsets. In *Proceedings of the Twenty-seventh Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, New York, NY, USA, 2015. ACM.

[MPX13] Gary L. Miller, Richard Peng, and Shen Chen Xu. Parallel graph decompositions using random shifts. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 196–203, New York, NY, USA, 2013. ACM.

[MR85] Gary L. Miller and John H. Reif. Parallel tree contraction and its application. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, SFCS '85, pages 478–489, Washington, DC, USA, 1985. IEEE Computer Society.

[MR89] Gary L. Miller and John H. Reif. Parallel tree contraction part 1: Fundamentals. In Silvio Micali, editor, *Randomness and Computation*, pages 47–72. JAI Press, Greenwich, Connecticut, 1989. Vol. 5.

[OSV12] Lorenzo Orecchia, Sushant Sachdeva, and Nisheeth K. Vishnoi. Approximating the exponential, the Lanczos method and an $\tilde{O}(m)$-time spectral algorithm for balanced separator. In *Proceedings of the 44th symposium on Theory of Computing*, STOC '12, pages 1141–1160, New York, NY, USA, 2012. ACM.

[Pet10] Seth Pettie. Distributed algorithms for ultrasparse spanners and linear size skeletons. *Distributed Computing*, 22(3):147–166, 2010.

[PR05] Seth Pettie and Vijaya Ramachandran. A shortest path algorithm for real-weighted undirected graphs. *SIAM Journal on Computing*, 34(6):1398–1431, 2005.

[PS89] David Peleg and Alejandro A. Schäffer. Graph spanners. *Journal of Graph Theory*, 13(1):99–116, 1989.

[PS14] Richard Peng and Daniel A. Spielman. An efficient parallel solver for SDD linear systems. In *Proceedings of the Forty-sixth Annual ACM Symposium on Theory of Computing*, STOC '14, pages 333–342, New York, NY, USA, 2014. ACM.

[PU89] David Peleg and Jeffrey D. Ullman. An optimal synchronizer for the hypercube. *SIAM J. Comput.*, 18(4):740–747, August 1989.

[Rei85] John H. Reif. An optimal parallel algorithm for integer sorting. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, SFCS '85, pages 496–504, Washington, DC, USA, 1985. IEEE Computer Society.

[Rei93] John H. Reif. *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1993.

[Saa03] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, second edition, 2003.

[SDB14] Julian Shun, Laxman Dhulipala, and Guy Blelloch. A simple and practical linear-work parallel algorithm for connectivity. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, pages 143–153, New York, NY, USA, 2014. ACM.

[She09] Jonah Sherman. Breaking the multicommodity flow barrier for $o(\sqrt{\log n})$-approximations to sparsest cut. In *Proceedings of the 2009 50th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '09, pages 363–372, Washington, DC, USA, 2009. IEEE Computer Society.

[She13] J. Sherman. Nearly maximum flows in nearly linear time. In *Foundations of Computer Science (FOCS), 2013 IEEE 54th Annual Symposium on*, pages 263–269, October 2013.

[SS99] Hanmao Shi and Thomas H. Spencer. Timework tradeoffs of the single-source shortest paths problem. *J. Algorithms*, 30(1):19–32, January 1999.

[SS11] Daniel A. Spielman and Nikhil Srivastava. Graph sparsification by effective resistances. *SIAM Journal on Computing*, 40(6):1913–1926, 2011.

[ST03] Daniel A. Spielman and Shang-Hua Teng. Solving sparse, symmetric, diagonally-dominant linear systems in time $o(m^{1.31})$. In *Proceedings of the 44th Annual IEEE*

Symposium on Foundations of Computer Science, FOCS '03, pages 416–427, Washington, DC, USA, October 2003. IEEE Computer Society.

[ST11] Daniel A. Spielman and Shang-Hua Teng. Spectral sparsification of graphs. *SIAM Journal on Computing*, 40(4):981–1025, 2011.

[ST13] Daniel A. Spielman and Shang-Hua Teng. A local clustering algorithm for massive graphs and its application to nearly linear time graph partitioning. *SIAM Journal on Computing*, 42(1):1–26, 2013.

[ST14] Daniel A. Spielman and Shang-Hua Teng. Nearly linear time algorithms for preconditioning and solving symmetric, diagonally dominant linear systems. *SIAM Journal on Matrix Analysis and Applications*, 35(3):835–885, 2014.

[SWT16] Veeru Sadhanala, Yu-Xiang Wang, and Ryan Tibshirani. Graph sparsification approaches for laplacian smoothing. In *Artificial Intelligence and Statistics*, pages 1250–1259, 2016.

[Tar79] Robert Endre Tarjan. Applications of path compression on balanced trees. *J. ACM*, 26(4):690–715, October 1979.

[Tho00] Mikkel Thorup. Floats, integers, and single source shortest paths. *J. Algorithms*, 35(2):189–201, May 2000.

[Tro12] Joel A. Tropp. User-friendly tail bounds for sums of random matrices. *Found. Comput. Math.*, 12(4):389–434, August 2012.

[TZ05] Mikkel Thorup and Uri Zwick. Approximate distance oracles. *J. ACM*, 52(1):1–24, January 2005.

[UY91] Jeffrey D. Ullman and Mihalis Yannakakis. High probability parallel transitive-closure algorithms. *SIAM J. Comput.*, 20(1):100–125, February 1991.

[Vai91] Pravin M. Vaidya. Solving linear equations with symmetric diagonally dominant matrices by constructing good preconditioners. A talk based on this manuscript was presented at the IMA Workshop on Graph Theory and Sparse Matrix Computation, October 1991.

[Vui80] Jean Vuillemin. A unifying look at data structures. *Commun. ACM*, 23(4):229–239, April 1980.