universität wien

Dissertation / Doctoral Thesis

Titel der Dissertation / Title of the Doctoral Thesis

Dynamic Graph Algorithms and Graph Sparsification: New Techniques and Connections

verfasst von / submitted by

Gramoz Goranci M. Sc.

angestrebter akademischer Grad / in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften (Dr. techn.)

Wien, 2019 / Vienna, 2019

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	A 786 880
Dissertationsgebiet lt. Studienblatt / field of study as it appears on the student record sheet:	Informatik
Betreuerin / Supervisor:	Univ.-Prof. Dr. Monika Henzinger

Abstract

Graphs naturally appear in several real-world contexts including social networks, the web network, and telecommunication networks. While the analysis and the understanding of graph structures have been a central area of study in algorithm design, the rapid increase of data sets over the last decades has posed new challenges for designing efficient algorithms that process large-scale graphs. These challenges arise from two usual assumptions in classical algorithm design, namely that graphs are static and that they fit into a single machine. However, in many application domains, graphs are subject to frequent changes over time, and their massive size makes them infeasible to be stored in the memory of a single machine.

Driven by the need to devise new tools for overcoming such challenges, this thesis focuses in two areas of modern algorithm design that directly deal with processing massive graphs, namely dynamic graph algorithms and graph sparsification. We develop new algorithmic techniques from both dynamic and sparsification perspective for a multitude of graph-based optimization problems which lie at the core of Spectral Graph Theory, Graph Partitioning and Metric Embeddings. Our algorithms are faster than any previous one and design smaller sparsifiers with better (approximation) quality. More importantly, this work introduces novel reduction techniques that show unexpected connections between seemingly different areas such as dynamic graph algorithms and graph sparsification. In particular we obtain the following results:

- The first dynamic algorithm for maintaining approximate solutions to Laplacian systems in sub-linear update and query time and an extension of the technique to dynamically maintaining variants of Vertex Spectral Sparsifiers and All-Pair Effective Resistances in undirected, weighted graphs. We also prove conditional lower bounds that certify that there are no efficient dynamic algorithms for maintaining Effective Resistances exactly.
- The first dynamic algorithm for maintaining low-stretch spanning trees with sub-polynomial stretch and sub-linear update time in undirected, unweighted graphs and an extension of the technique to dynamically maintaining low-diameter clustering.
- The current best-known algorithms for incrementally maintaining global Minimum Cut, approximate All-Pair Maximum Flow and Sparsest Cut in undirected, unweighted graphs. A key primitive behind our algorithms is a new notion of Local Sparsifiers, a stronger variant of the well-studied notion of Vertex Sparsifiers.
- The current best-known algorithm for constructing vertex sparsifiers that are minors of the input graph and preserve shortest path distances approximately and reachability information exactly. We derive upper-bounds on the quality and size of such sparsifiers and also prove lower-bounds that better explain the trade-off between these two quantities.

Zusammenfassung

Graphen sind passende Modelle in mehreren realen Kontexten, unter anderem in sozialen Netzwerken, dem Web-Netzwerk und in Telekommunikationsnetzen. Die Analyse und das Verständnis von Graphstrukturen sind ein zentraler Gesichtspunkt im Design von Algorithmen. Jedoch stellt das rasante Wachstum an Datenmengen neue Herausforderungen an das Design von effizienten Algorithmen für riesige Graphen. Diese Herausforderungen entspringen aus zwei Annahmen des klassischen Algorithmendesigns, und zwar dass Graphen statisch sind und in den Speicher einer einzelnen Machine passen. Jedoch sind Graphen in vielen Anwendungen in konstanter Veränderung und oftmals zu groß, um im Speicher einer einzelnen Maschine gespeichert werden zu können.

Getrieben durch den Bedarf, neue Lösungen für diese Herausforderungen zu finden, fokussiert sich diese Dissertation auf zwei Bereiche des modernen Algorithmendesigns, um Lösungen für diese Probleme zu finden; nämlich dynamische Graphalgorithmen und Graphsparsifikation. Wir entwickeln neue algorithmische Techniken für beide Bereiche, um graphbasierte Optimierungsprobleme unter anderem in Spectral Graph Theory, Graph Partitioning und Metric Embeddings effizienter lösen zu können. Unsere Algorithmen sind schneller als jegliche vorherige und wir entwickeln kleinere Sparsifier mit besserer Approximationsqualität. Außerdem entwickelt diese Arbeit neuartige Reduktionstechniken, welche unerwartete Zusammenhänge zwischen scheinbar verschiedenen Bereichen, wie zum Beispiel dynamische Graphalgorithmen und Graphsparsifikation aufzeigen, insbesondere erreichen wir die folgende Resultate:

- Den ersten dynamischen Algorithmus für die Aufrechterhaltung von approximativen Lösungen für Laplacian Systeme mit sub-linearer Update und Query Time und eine Erweiterung der Technik, um dynamisch Vertex Spectral Sparsifiers und All-Pair Effective Resistances in ungerichteten, gewichteten Graphen aufrechtzuerhalten. Wir beweisen außerdem Conditional Lower Bounds, welche beweisen, dass es keinen effizienten dynamischen Algorithmus geben kann, der Effective Resistances exakt berechnet.
- Den ersten dynamischen Algorithmus, der Low-stretch aufspannende Bäume mit sub-polynomialem Stretch und sub-lineare Update Time in ungerichteten, ungewichteten Graphen aufrecht erhält. Außerdem eine Erweiterung der Technik, um dynamische Cluster mit niedrigem Diameter aufrechtzuerhalten.
- Den besten bekannten Algorithmus für inkrementellen global Minimum Cut, approximativen All-Pair Maximum Flow und Sparsest Cut in ungerichteten, ungewichteten Graphen. Ein wichtiger Baustein für diese Algorithmen ist ein neuentwickeltes Konzept namens Local Sparsifier, eine stärkere Variante der bekannten Vertex Sparsifier.
- Den besten bekannten Algorithmus für die Konstruktion von Vertex Sparsifiers, die Minors des Inputgraphen sind und eine Approximation der kürzesten Wege und die exakte Erreichbarkeit aufrecht erhalten. Wir entwickeln obere

Schranken für die Qualität und Größe solcher Sparsifier und beweisen untere Schranken, welche den Kompromiss der beiden Zielfunktionen besser erklären.

Acknowledgments

First and foremost, I would like to express my sincerest gratitude to my advisor Monika Henzinger, for her guidance and patience over these last four years of my Ph.D. She has always been there to listen to my ideas, generously shared her research insights with me and encouraged me to work on topics that I was most excited about. She gave never-ending support, especially when things did not go smoothly. She has been, and continues to be, a great source of inspiration for me, both in terms of academic and personal development.

I am deeply thankful to Harald Räcke, for being a great mentor and collaborator, and for hosting me several times at TU Munich. His approach and intuition to problem solving continue to amaze me, and I feel fortunate to have learned from his insights.

I am particularly grateful to Richard Peng, who hosted me during my research stay at Georgia Tech in Atlanta. I have benefited immensely from his generosity, research insights in numerical and graph algorithms, his academic advice, guidance and our joint collaborations.

Special thanks to Robert Krauthgamer and Danupon Nanongkai who have agreed to review this thesis and be part of my thesis committee. I have enjoyed and gained a lot from research conversations with both of them. I would also like to thank Thatchaphol Saranurak and Mikkel Thorup for the remarkable research collaborations.

It has been a great experience to spend these years as part of the TAA group. I want to thank Sebastian for being a great office mate, for the collaboration and the academic advise; Pan for all the collaboration and exceptional support; Darek for working with me, being an inspiring friend and for playing music together; Marco for being the co-author of my first paper; Veronika for being a caring colleague and convincing me to come to Vienna for my Ph.D.; Stefan for being there throughout our joint path and for all the conversations on life; Wolfgang for his support, and for organizing several entertaining non-academic events; Alex for being a kind and cheerful office mate; Valon and Labinot for the numerous conservations about life and politics during lunches at Mensa; Ulli, Birgit and Christina for keeping the administrative workload low and allowing me to practice my German with them, former and current colleagues, and visitors for the pleasant atmosphere they have created. I have had a very rewarding experience during my visit in Atlanta, and for this I thank David, Saurabh, Yu, Matthew and Di for being great friends and colleagues.

I am indebted to the endless support and love my family has given me during these years of graduate school. I thank my parents, Suzana and Kamuran, my siblings Gladiola and Shkamb and their spouses Labi and Ida, and my in-laws Shpresa, Nail and Pellumb, thank you all for being there for me! A special thank you goes to my grandparents, especially to my late grandfather Rifat, who was a great source of inspiration for me and infused within me the passion for science.

Finally, I would like to wholeheartedly thank my fiancè Edona for her love, support and for cheering me up (especially after conference notifications), and reminding me often that life is a strict superset of research. Thank you for also proof-reading many of my manuscripts, grant proposals and parts of this thesis.

Bibliographic Note

Most of the results of this thesis were already published in conference proceedings and journal articles. Therefore, the chapters of this thesis are based on the following publications and manuscripts:

- **Chapter 2:** Gramoz Goranci, Monika Henzinger, and Pan Peng. "Dynamic Effective Resistances and Approximate Schur Complement on Separable Graphs". In: *European Symposium on Algorithms (ESA)*. 2018, 40:1–40:15.
- **Chapter 3:** David Durfee, Yu Gao, Gramoz Goranci, and Richard Peng. "Fully Dynamic Spectral Vertex Sparsifiers and Applications". In: *Symposium on Theory of Computing (STOC)*. (forthcoming). 2019.
- **Chapter 4:** Sebastian Forster and Gramoz Goranci. "Dynamic Low-Stretch Trees via Dynamic Low-Diameter Decompositions". In: *Symposium on Theory of Computing (STOC)*. (forthcoming). 2019.
- **Chapter 5:** Gramoz Goranci, Monika Henzinger, and Mikkel Thorup. "Incremental Exact Min-Cut in Polylogarithmic Amortized Update Time". In: *ACM Trans. Algorithms* 14.2 (2018). Announced at ESA'16, 17:1–17:21.
- **Chapter 6:** Gramoz Goranci, Monika Henzinger, and Thatchaphol Saranurak. "Fast Incremental Algorithms via Local Sparsifiers". manuscript. 2019.
- **Chapter 7:** Yun Kuen Cheung, Gramoz Goranci, and Monika Henzinger. "Graph Minors for Preserving Terminal Distances Approximately - Lower and Upper Bounds". In: *International Colloquium on Automata Languages and Programming (ICALP)*. 2016, 131:1–131:14.
- **Chapter 8:** Gramoz Goranci, Monika Henzinger, and Pan Peng. "Improved Guarantees for Vertex Sparsification in Planar Graphs". In: *European Symposium on Algorithms (ESA)*. 2017, 44:1–44:14.

Contents

CHAPTER 1

Introduction

Recent technological developments, in particular the increased popularity of the Internet, have lead to an enormous increase in data volumes. While the access to this large amount of data has allowed us to gain complex insights and identify various patterns about data, performing basic computational tasks and storing the data have posed major challenges that require new treatments beyond the usage of traditional applications and tools. This leaves computer scientists the mandate to address such challenges by developing models that better suit the modern technological advances.

A considerable fraction of the generated data can be modeled using *Graphs*. A graph is a collection of nodes and a collection of edges, where each edge connects a pair of nodes. Graphs are ubiquitous structures in mathematics and computer science, and naturally appear in several real-world contexts including social networks (e.g., the Facebook graph), the web network, and telecommunication networks. In comparison to other data representations, graphs are particularly desirable since (suitably) visualizing them often offers ways to identify interesting patterns in the data, e.g., detecting communities in social networks. Another reason why graph representations are found appealing is because algorithms for manipulating and storing them have been thoroughly studied since the early days of algorithm design. Nevertheless, a large number of these graph algorithms work under the assumptions that graphs are static, i.e., that they do not undergo changes and that they can be stored into the memory of a single machine. Unfortunately, these assumptions fail to capture graphs that appear in many important real-world scenarios.

As a motivating example consider a map graph, where each node corresponds to a city and an edge between any two nodes represents the route connecting them. This map graph also includes the length of each route by labeling the edges with the corresponding distance. A fundamental question in algorithm design is to understand the metric structure of the map graph; more concretely, we want to compute the shortest path distance between any two given nodes in the map graph. This task

has been addressed by many classical algorithms and the running time complexity of this problem is shown to be cubic on the number of nodes in the graph. However, it does not take too much effort to realize that real-world map graphs undergo changes. For example, due to construction work, it may occur that some road connecting two cities is blocked, which in turn implies that the edge connecting these cities is deleted from the map graph. The deletion of such an edge might affect the shortest path between cities, thus implying that the old solution is incorrect for the new map graph. One obvious way to correct the solution is to recompute shortest paths from scratch in the new graph. However, this trivial solution comes at the expense of high computational burden, which is not feasible for small devices with limited resources, e.g., navigation systems. A set of natural questions that arise are the following: Can we design methods that perform better than re-computation from scratch? If yes, what is the best possible speed-up that we can achieve? Do we have to pay in the quality of the solution to get better performance?

Another common challenge that we face when dealing with huge graphs are computational and storage resources. This is due to the fact that the size of a graph can be as large as quadratic in the number of nodes. A traditional approach to address this issue has been to *compress* large graphs into smaller ones while preserving properties or features of interest. These compressed versions of graphs are particularly desirable since any computational task on the original graph can be now performed on the compressed graph, thus leading to significant savings in computational and storage resources. Graph compression is commonly studied from two perspectives: (1) reducing the number of edges of a graph, (2) and reducing the number of nodes. While the first approach has been successfully employed for improving the running time of many basic graph problems, its practical applicability is somewhat limited due to the fact that most large networks are already sparse. As a result, compression tools that reduce the number of nodes have received increasing attention over the last decade. To illustrate, let us go back to the graph map example. Suppose that among all cities in the graph, we are interested only in a subset of nodes that are "important" to us. This is relevant in many practical scenarios, e.g., one desires to preserve distance information only among big cities while ignoring the small ones. The following questions naturally arise: Can we compress the map graph into a graph only on the big cities while preserving distances? What is the incurred loss when transferring from the large map graph to the smaller one? What is the trade-off between the loss and the size of the compressed graph?

All of the above questions and their variants will be addressed in this thesis. In particular, we will provide provable algorithmic tools from both dynamic and compression perspective for a multitude of graph-based optimization problems that arise in Spectral Graph Theory, Graph Partitioning and Metric Embeddings. More importantly, this thesis establishes novel reduction techniques that reveal unexpected connections between time-evolving graphs and graph compression. In what follows, we will first review results in dynamic graph algorithms and then discuss our contributions in the area of graph sparsification.

1.1 Dynamic Graph Algorithms

Suppose we are given a graph $G = (V, E)$ and a property $\mathcal{P}$ with respect to G. Furthermore, assume that the structure of G is slightly perturbed, that is, an edge is either inserted or deleted from G. Can we efficiently update the property $\mathcal{P}$ in the perturbed graph rather than recomputing it from scratch? This basic question has been asked for many important graph properties for decades and the area that exclusively studies these questions is called *Dynamic Graph Algorithms*.

More concretely, a *dynamic graph algorithm* is a data structure that supports the following operations on a given input graph G:

- PREPROCESS(G): preprocess the graph G
- INSERT(u, v): insert the edge (u, v) to G
- DELETE(u, v): delete the edge (u, v) from G
- QUERY($\mathcal{P}$): query the property $\mathcal{P}$

In some variants of dynamic graph algorithms, the query operation might not be supported and the goal there is simply to maintain a correct property $\mathcal{P}$ with respect to the current graph at any point in time. A dynamic algorithm is characterized with three different time measures: (1) *processing time*, which denotes the time to support operation PREPROCESS(G); (2) *update time*, which denotes the time to support operations INSERT(u, v) and DELETE(u, v), and (3) *query time*, which denotes the time to support operation QUERY($\mathcal{P}$). Update and query times can either be *worst-case*, that is, the time spent to process each update or query individually, or *amortized*, that is, the running time amortized over a sequence of operations.

Depending on the types of update operations we support, dynamic algorithms are classified into three main categories: (i) *fully dynamic*, if update operations consist of both edge insertions and deletions; (ii) *incremental*, if update operations consist of edge insertions only; and (iii) *decremental*, if update operations consist of edge deletions only. When studying the update times in algorithms of type (ii) and (iii), it is common to consider *total update time*, which is the time spent over a sequence of $\Theta(m)$ insertions or deletions, where m denotes the number of final or initial edges in the graph, respectively. Dynamic algorithms can either be *deterministic* or *randomized*, and usually algorithms with better running times are obtained if randomization is allowed. A common assumption in randomized dynamic algorithm is that the adversary is *oblivious*, that is, the sequence of updates and queries is fixed in advance by the adversary, and the choices are are revealed to the algorithm one by one.

There has been outstanding progress on devising efficient dynamic graph algorithms, especially during the last two decades, and the graph properties that have been considered include connectivity [129, 130, 135, 147, 249], reachability [62, 80, 131, 164, 217, 219, 221, 223], shortest paths [41, 46, 79, 132–134, 247], matching [31, 47, 48, 123, 204, 206], (global) minimum cut [127, 152, 246, 248], minimum spanning tree [106, 128, 138], spanner [27, 33, 45, 54, 90], cut and spectral sparsifier [12],

etc. However, despite this volume of work, there is a large number of questions that remain poorly understood. The situation is even worse if we consider several "non-basic" graph problems e.g., variants of graph partitioning, where no non-trivial solutions are known. Driven by this, in this thesis we study dynamic algorithms for new graph properties that appear to be important in different application domains but have not been considered so far. We also make progress on fundamental basic graph problems by improving their long-standing running time guarantees.

1.1.1 Dynamic Algorithms for Spectral Primitives

In this thesis we study algorithms for dynamically maintaining solutions to Laplacian systems and Effective Reistances (see Chapters 2 and 3). Laplacian systems are an important subclass of linear systems which arise in many natural contexts and have found applications in machine learning, computer graphics and image processing. Solving Laplacian systems has received considerable attention after the seminal work of Spielman and Teng [237] who devised the first near-linear time solver. A formal definition of such systems is given below.

Given a graph $G = (V, E)$ with n nodes and m edges, let $\mathbf{L} := \mathbf{D} - \mathbf{A}$ denote the *Laplacian matrix* of G, where $\mathbf{D}$ and $\mathbf{A}$ are the associated degree and adjacency matrix of G, respectively. Matrix $\mathbf{L}$ together with a vector $\mathbf{b} \in \mathbb{R}^n$ form a system of linear equations $\mathbf{Lx} = \mathbf{b}$, which is referred to as *Laplacian system*. Let $\mathbf{L}^\dagger$ denote the pseudo-inverse of $\mathbf{L}$. The *solution vector* $\mathbf{x} \in \mathbb{R}^n$ satisfies $\mathbf{x} = \mathbf{L}^\dagger\mathbf{b}$, and it exists if and only if $\mathbf{b}^\top\mathbf{1} = 0$, where $\mathbf{1}$ is the all-ones n-dimensional vector. Let $\mathbf{1}_u \in \mathbb{R}^n$ denote the indicator vector of a vertex u such that $\mathbf{1}_u(v) = 1$ if $v = u$ and 0 otherwise.

We introduce a dynamic model for solving Laplacian systems that supports insertions and deletions of edges in the underlying graph (which correspond to modifying entries in $\mathbf{L}$), modifications to vector $\mathbf{b}$, and query access to one or few coordinates of an approximate solution vector, all in sublinear time. Concretely, we obtain the following result.

Theorem 1.1.1. *Given any error parameter $\epsilon \in (1/m, 1)$, there is a fully-dynamic algorithm for solving Laplacian systems on undirected, unweighted bounded-degree graphs while supporting insertions and deletions of edges, modifications to vector $\mathbf{b}$, as well as query access to one or few entries of a vector $\tilde{\mathbf{x}}$ such that $\left\|\tilde{\mathbf{x}} - \mathbf{L}^\dagger\mathbf{b}\right\|_{\mathbf{L}} \leq \epsilon \left\|\mathbf{L}^\dagger\mathbf{b}\right\|_{\mathbf{L}}$, all in $\tilde{O}(n^{11/12}\epsilon^{-5})$[1] expected amortized time. These guarantees hold against an oblivious adversary.*

A spectral primitive closely related to Laplacian systems is effective resistance, a graph property that has received increasing attention recently due to its application in speeding up algorithms for several cornerstone graph problems [70, 187, 189, 227].

[1]Throughout this thesis, we use $\tilde{O}(\cdot)$ to hide polylogarithmic factors, i.e., $\tilde{O}(f(n)) = O(f(n) \cdot \text{poly}(\log f(n)))$.

Given a graph $G = (V, E, \mathbf{w})$ with $\mathbf{w}$ assigning non-negative weights to edges in E, and any pair of vertices $u, v \in V$, we let

$$R^G_{\text{eff}}(u, v) := (\mathbf{1}_u - \mathbf{1}_v)^\top \mathbf{L}^\dagger (\mathbf{1}_u - \mathbf{1}_v)$$

denote the *effective resistance* between u and v in G. When G is viewed as a resistor network, where resistances are the inverse of the edge weights, effective resistance between u and v can be thought of as the energy of the flow when routing one unit of current from u to v.

We study fully-dynamic graph algorithms for maintaining All-Pair Effective Resistances. Surprisingly enough, we show that this graph property admits sub-linear update and query times while achieving very high approximation accuracy to effective resistance. This is in stark contrast to related graph measures like shortest path, for which (conditional) hardness results are known in the fully-dynamic setting [135], and maximum flow, which remains poorly understand from the dynamic perspective.

Theorem 1.1.2. *For any given error parameter $\epsilon \in (0, 1)$, there is a fully-dynamic algorithm for maintaining $(1 \pm \epsilon)$-approximation to effective resistances in undirected, unweighted graphs while supporting insertions and deletions of edges as well as pairwise effective resistance queries, all in $\tilde{O}\left(\min\{m^{3/4}, n^{5/6}\}\epsilon^{-4}\right)$ expected amortized time. Our guarantees hold against an oblivious adversary.*

We extend the above result in two directions. First, the above algorithm can be extended to also handle weighted graphs, albeit with a bound of $\tilde{O}(n^{5/6}\epsilon^{-4})$ on the expected amortized update and query time. Second, if we restrict to weighted graphs that admit *small* separators, e.g., planar graphs, our *worst-case* running time guarantees improve to $\tilde{O}(\sqrt{n}\epsilon^{-2})$. The key idea behind Theorem 1.1.2 and its corresponding extensions is dynamically maintaining an approximation to *Schur complements* (also known as *vertex spectral sparsifiers*). Roughly speaking, given a graph G and a subset of vertices K, a Schur complement is a graph with vertex set K that preserves effective resistances among any pair of vertices from K in G.

Despite the fact that our results share the same idea at a high level, there are subtle differences between their implementations. For general graphs, our techniques crucially rely on the fact that Schur complements can be viewed as a sum of random walks. This allows us to subsample vertices from the original graph and then construct a Schur complement with respect to this subsampled vertex set. The subsampling makes sure that the random walks are short and thus they can be maintained dynamically using elementary data-structures. On the other hand, for planar graphs we exploit the fact that they admit sub-linear separators (hence the $\sqrt{n}$ dependency on the running time), as well as the fact that approximate Schur complements can be computed in nearly-linear time [88]. Inspired by the seminal work of Lipton, Rose and Tarjan [185] on *nested dissection*, these two ingredients are then brought together to dynamically maintain Schur complements for this family of graphs.

All results we presented above guarantee only approximate answers to effective resistance queries. An obvious question is whether there are dynamic algorithms that can *exactly* report effective resistances while still achieving sub-linear update and query time. We show that this is likely not the case. In particular, assuming a certain believable conjecture, we prove that there are no algorithms that simultaneously achieve sub-linear update and query time.

Theorem 1.1.3. *No incremental or decremental algorithm can maintain the (exact) (s,t) effective resistance in general graphs on n vertices with both $O(n^{1-\delta})$ worst-case update time and $\tilde{O}(n^{2-\delta})$ worst-case query time for any $\delta > 0$, unless the OMv conjecture [135] is false.*

The preceding result can be extended to graphs that admit small separators, albeit with guarantees of $O(n^{1/2-\delta})$ and $O(n^{1-\delta})$ on the update and query time, respectively. At the heart of our reductions, that prove these results, is a relation between effective resistance and the problem of detecting cycles of certain length in a graph. We defer the reader to Chapter 2 for more details.

1.1.2 Dynamic Low-Stretch Trees

In this thesis we study algorithms for dynamically maintaining Low-Stretch Spanning Trees and Spanners (see Chapter 4). Trees are the simplest class of graphs. From the algorithmic point of view, they are very appealing since many graph-based problems admit somewhat easier solutions when restricted to tree instances. In order to be able to exploit such a desirable behavior of trees, the problem of approximating general graphs by trees while preserving relevant graph properties, has been extensively studied in algorithm design. One notable example is Low-Stretch Spanning Tree, which at a high level is a spanning tree of a given input graph that preserves distances on average with a small stretch. Such trees are a central concept in Metric Embeddings and have found numerous applications in fast solvers for symmetric diagonally dominant (SDD) linear systems [161], in the construction of competitive oblivious routing schemes [214] and in approximation algorithms [207]. A formal definition of low-stretch trees is given below.

Given a graph $G = (V, E, \mathbf{w})$ and any $u, v \in V$, let $\text{dist}_G(u, v)$ denote the length of a shortest path between u and v in G. Let T be a spanning tree of G. We define the *stretch* of an edge $(u, v) \in E$ with respect to T to be $\text{stretch}_T(u, v) := \frac{\text{dist}_T(u,v)}{\mathbf{w}(u,v)}$. The *average stretch* over all edges of G with respect to T is given by

$$\text{avg-stretch}_T(G) := \frac{1}{|E|} \sum_{(u,v)\in E} \text{stretch}_T(u, v).$$

We say that T is a *low-stretch* spanning tree whenever the average stretch is sub-polynomial or poly-logarithmic in the number of nodes $n = |V|$.

Motivated by the fundamental importance of low-stretch spanning trees as well as their powerful applications, we considered the maintenance of this object from

a dynamic point of view. Indeed, designing dynamic algorithm for such trees was posed as an open problem by Baswana et al. [33]. However, despite the extensive research in dynamic algorithms in recent years, no progress was made in this direction. In this thesis, we show the first non-trivial guarantees for this problem.

Theorem 1.1.4. *Given any unweighted, undirected graph with n nodes undergoing edge insertions and deletions, there is a fully dynamic algorithm for maintaining a spanning tree of expected average stretch $n^{o(1)}$ that has expected amortized update time $n^{1/2+o(1)}$. These guarantees hold against an oblivious adversary.*

The above algorithm can be slightly modified to give average stretch $O(t)$ and update time $n^{1+o(1)}/t$ for $t \geq \sqrt{n}$. This shows that the $\sqrt{n}$ barrier in the running time is not inherent, at least if a very large stretch is tolerable. One of the major building blocks of our algorithm is to dynamically maintain a clustering of a graph into small-diameter clusters (also known as *low-diameter decomposition*). This is implemented using the random-shift clustering due to Miller, Peng and Xu [195] together with many adaptations to make it work in the dynamic setting. We then employ a dynamic version of the hierarchy of low-diameter clusters due to Alon, Karp, Peleg, and West [17], which in turn requires a sophisticated amortization approach to control propagation of updates within the hierarchy. Additionally our algorithm uses dynamic cut sparsifiers to reduce the problem to sparse graphs. While it is known that cuts and distances are dual to each other in similar settings [23], our argument requires a slight deviation from common approaches.

The dynamic random-shift clustering could be of independent interest. Indeed, a direct consequence of this technique improves the previously best-known guarantees for dynamically maintaining graph spanners. Roughly speaking, a *graph spanner* is a (sparse) subgraph of a given graph G that preserves all pair shortest path distances of G up to a multiplicative error.

Theorem 1.1.5. *Let $t \geq 1$ be a parameter. Given any unweighted, undirected graph with n nodes undergoing edge insertions and deletions, there is a fully dynamic algorithm for maintaining a spanner of stretch $(2t-1)$ and expected size $O(n^{1+1/t}\log n)$ that has expected amortized update time $O(t\log^2 n)$. These guarantees hold against an oblivious adversary.*

Compared to the state-of-the art result of Baswana et al. [33], the above theorem improves upon the size of the spanner and the update time by a factor of t. Independently of our work, Saranurak and Wang [225] obtained similar guarantees for dynamically maintaining low-diameter clusters using different techniques. Concretely, they employ expander decomposition as a subroutine and use pruning to maintain a valid decomposition under edge deletions. We believe that our solution is arguably simpler than their expander pruning approach.

1.1.3 Dynamic Graph Partitioning

In this thesis we study incremental algorithms for maintaining Global Minimum Cut and Sparsest Cut, both being core concepts in Graph Partitioning (see Chapters 5 and 6). Graph partitioning problems typically involve partitioning the input graph into smaller components while minimizing the number of connections between these components. These problems have historically occupied a central place in understanding network flows [104], packet routing [209] and VLSI layout. They have also been employed in many divide-and-conquer approaches for solving clustering problems. In what follows we start by defining the global minimum cut problem and then later discuss the results related to the sparsest cut problem.

Given an unweighted, undirected graph $G = (V, E)$, and a subset of vertices $S \subseteq V$, the *edge cut* $E(S, V \setminus S)$ is a set of edges that have one endpoint in S and the other in $V \setminus S$. Let $\lambda(S) = |E(S, V \setminus S)|$ denote the size of the edge cut. A *global minimum cut* is a subset S whose edge cut size is the smallest among all subsets of vertices in G. Let $\lambda(G)$ denote the edge cut size of the global minimum cut in G. There has been extensive work on designing algorithms for computing global minimum cuts in the static setting and it is known that the problem can be solved in nearly linear-time [136, 150, 157].

The first work on dynamic Global Minimum Cut is due to Karger [152], who gave the first non-trivial running time guarantees for the problem. When both insertions and deletions of edges are supported, Thorup [248] achieves a $(1 + o(1))$ approximation to the value of global minimum cut in $\tilde{O}(\sqrt{n})$ update and query time, and these bounds are the best-known to date. However, none of these works applies to maintain the exact value of global minimum cut and Thorup [248] even poses this question as an open problem. One exception here is the work by Henzinger [127], who obtains an exact incremental algorithm with $\tilde{O}(\lambda(G))$ amortized update time, where $\lambda(G)$ is the value of the global minimum cut in the graph after all insertions are processed. Note that $\lambda(G)$ can be as large as $O(n)$ and the main question is whether a truly sub-linear running time can be achieved. In the following result, we show that this is indeed the case by providing an exponential speed up on the update time of Henzinger [127].

Theorem 1.1.6. *Given any unweighted, undirected graph with n nodes undergoing edge insertions, there is a deterministic incremental algorithm for exactly maintaining the value of a global minimum cut $\lambda(G)$ in $O(\log^3 n \log\log n)$ amortized time and $O(1)$ query time.*

The above result stays in sharp contrast to a *polynomial* conditional lower-bound for the fully dynamic *weighted* global minimum cut problem due to Nanongkai and Saranurak [201]. The high-level idea behind our result is to combine a sparsification routine of Kawarabayshi and Thorup [157] or its recent improvement by Henzinger, Rao and Wang [136], and an exact incremental algorithm of Henzinger [127]. We remark that the combination itself is not immediate and it entails opening the black-

boxes used in these works and skillfully extending them to obtain our desirable guarantees.

Motivated by the recent work on *space-efficient* dynamic algorithms [49], we also consider efficient maintenance of global minimum cut using only $\tilde{O}(n)$ space. The results we obtain achieve $O(n \log n)$ space while still being able to support insertions and (approximate) queries in poly-logarithmic and constant time, respectively. Note that this setting differs from the standard *graph stream model*, which typically allows $\tilde{O}(n)$ space while ignoring relevant measures like update and query time.

We next discuss our contribution related to the Sparsest Cut problem, which is a well-studied NP-hard problem, that often serves as a prime example when discussing applications of metric embeddings in combinatorial optimization. Given an unweighted, undirected graph $G = (V, E)$, and a subset of vertices $S \subseteq V$, we define the *uniform sparsity* of the *cut* $(S, V \setminus S)$ as $\Phi_G(S) := \frac{E_G(S, V \setminus S)}{|S| \cdot |V \setminus S|}$. The *uniform sparsest cut of* G is the cut $(S, V \setminus S)$ with smallest possible sparsity. Let $\Phi(G)$ denote the value of the sparsest cut in G. In the literature, there are several efficient algorithms for approximating $\Phi(G)$ with a multiplicative factor of $O(\log^c n)$, where $c \in [1/2, 1]$ [26, 163, 230]. However, prior to our work, nothing was known about the complexity of this problem in the dynamic setting.

We make the first positive progress towards understanding the Uniform Sparsest Cut problem from the dynamic point of view. In the insertions-only model, we show that we can maintain a poly-logarithmic approximation to the sparsest cut in sub-linear update time. As a by-product of our techniques, our algorithm provides a trade-off between the approximation error and the update time.

Theorem 1.1.7. *Let $t \geq 1$ be a parameter. Given any unweighted, undirected graph with n nodes undergoing edge insertions, there is a randomized incremental algorithm for maintaining an $O(\log^{8t} n)$ approximation to the value of uniform sparsest cut $\Phi(G)$ in $\tilde{O}(n^{2/(t+1)})$ worst-case update time time and $O(1)$ query time. Our algorithm extends to weighted graphs with polynomially bounded weights.*

The key idea behind the proof of Theorem 1.1.7 is a new notion of sparsifiers, called *local sparsifiers*. These sparsifiers are a stronger version of the well-studied notion of *vertex sparsifiers*. Concretely, in Vertex Sparsification, given a graph $G = (V, E)$ and a subset of vertices K, referred to as *terminals*, the goal is to construct a graph $H = (V', E')$ with $V' \supseteq K$ and $|V(H)|$ is "small" such that H preservers some graph property $\mathcal{P}$ that involves the terminals K in G (see the next section for an in-depth treatment on vertex sparsifiers). A *local sparsifier* is a data-structure generalization of vertex sparsifiers; formally, given a graph $G = (V, E)$, the goal is to build a data-structure that supports the following operations:

1. PREPROCESS(G): preprocess the graph G
2. QUERYSPARSIFIER(G, K): compute and output a vertex sparsifier H of G that preserves some property $\mathcal{P}$ among vertices in K.

In other words, this definition suggests that local sparsifiers allow us to extract vertex sparsifiers for any set of terminals in K. Note that operation (2) is a very strong requirement, as there are $\Theta(2^n)$ different terminal sets.

We show that a variant of *tree cut sparsifiers* due to Peng, Räcke, Shah and Täubig [210, 216] can be used to construct local sparsifier that preserve cut-structure of the graph up to poly-logarithmic factors while achieving $\tilde{O}(m)$ preprocessing time and $\tilde{O}(|K|)$ query time. In particular, this implies that the uniform sparsest cuts are also preserved within the same approximation. We then design a reduction that converts such an efficient local sparsifier into an incremental algorithm that maintains a poly-logarithmic approximation to the uniform sparsest cut. The same technique allows us to obtain very fast incremental algorithms for the approximate All-Pair Maximum Flow problem with similar guarantees to those in Theorem 1.1.7. This is quite intriguing since nothing was known about the dynamic Max-Flow problem in general graphs, even when allowing poly-logarithmic approximation.

In fact, our reduction relating local sparsifiers and incremental graph algorithms applies to a larger family of graph properties. For example, using variants of the *distance oracle* due to Thorup and Zwick [251] we construct efficient local sparsifiers, which in turn imply a *deterministic* incremental algorithm that approximates All-Pair Shortest Paths up to a constant factor in sub-linear update and query time.

Another important problem in dynamic algorithms is to understand the complexity of $(1 + \epsilon)$-approximate maximum flow problem in the dynamic setting. Even when restricted to the weaker *offline dynamic* model, where edge updates and queries are given in advance, the problem remains poorly understood. On the other hand, over the last years there has been increasing interest in proving conditional polynomial lower-bounds for dynamic problems. A property that most of these lower-bounds share is that they apply to the offline dynamic model. Driven by this, we develop a framework that connects offline dynamic problems and vertex sparsification. Specifically, we show that if there are efficient $(1 + \epsilon)$ vertex sparsifiers of size $\tilde{O}(\text{poly}(|K|, 1/\epsilon))$ that preserve cuts, then the approximate offline maximum flow problem admits sub-linear update and query times. This would imply that no $\Omega(n^{1-o(1)})$ lower bound can be shown for the approximate offline max flow problem. For other connections we refer the reader to Chapter 6.

1.2 Graph Sparsification

A *graph sparsifier* is a "compressed" version of a large input graph that preserves properties like distance or reachability information, cut value or graph spectrum. Traditionally, graph sparsifiers have been studied from two perspectives: (1) those that reduce the number of edges of a graph, referred to as *edge sparsifiers*, and (2) those that reduce the number of nodes, referred to as *vertex sparsifier*. Edge sparsifiers have been successfully applied for improving the running time of many basic graph-based optimization problems, and the most notable examples include transitive reductions [15], spanners [20], cut sparsifiers [35] and spectral sparsifier [238].

In this thesis, we focus on vertex sparsifiers. Concretely, given a graph $G = (V, E)$ and a subset of vertices K, referred to as *terminals*, the goal is to construct a graph $H = (V', E')$ satisfying the following properties:

- $V' \supseteq K$ and $|V'|$ is "small", ideally $|V'| = O(\text{poly}(|K|))$,
- H (approximately) preserves properties like reachability, distance, cuts or multi-commodity flows defined among terminals in K; often it is desirable that H is structurally similar to G, e.g., when G is planar, so is H

When H preserves some property approximately, the approximation ratio is referred to as the *quality* of the sparsifier. The usefulness of such a sparsification tool is apparent from an algorithm point of view; once H is computed, we can perform algorithmic tasks only in H instead of G, which in turn leads to savings in computational and storage resources. Besides their practical relevance, vertex sparsifiers have also found applications within other sub-areas of Theoretical Computer Science, namely approximation algorithms [94, 197], dynamic graph algorithms [113, 115], and network routing [73].

In what follows, which constitutes Chapters 7 and 8 of this thesis, we will discuss our contributions on vertex sparisifers that are structurally similar to the input graph and at the same time preserve distances in undirected graphs or reachability information in directed graphs.

1.2.1 Distance Approximating Minors

We study vertex sparsifiers that are obtained using minor operations while preserving distances among terminal pairs approximately. Minors are particularly desirable since they preserve structural properties of the input graph, e.g., a minor of a planar graph is another planar graph. Formally, given a weighted graph $G = (V, E, \mathbf{w})$ and a designated subset of terminals K, an *α-distance approximating minor* of G is a weighted graph $H = (V', E', \mathbf{w}')$ such that

- $V' \subseteq K$ and V' is small, ideally $|V'| = O(\text{poly}(|K|))$,
- H is a minor of G, i.e., H is obtained from G by deleting edges and vertices any by contracting edges. No terminal can be deleted, and no two terminals can be contracted together.
- Terminal distances are preserved up to an α factor, i.e., for any pair of vertices $u, v \in K$, we have

$$\text{dist}_G(u, v) \leq \text{dist}_H(u, v) \leq \alpha \cdot \text{dist}_G(u, v).$$

Vertices in $V' \setminus K$ are usually referred to as *non-terminals* or *Steiner vertices*. Gupta [119] introduced the strongest version of the problem which requires that $V' = K$, also known as the *Steiner Removal Problem*. In this setting, he showed that trees admit sparsifiers with quality 8. Kamma, Krauthgamer and Nguyen [146]

showed that general graphs admit sparsifiers with quality $O(\log^5(|K|))$. This bound has been subsequently improved to $O(\log^2(|K|))$ by Cheung [68] and finally to $O(\log(|K|))$ by Filtser [102].

At the other extreme, Krauthgamer, Nguyen and Zondiner [170] considered the setting where distances are preserved *exactly*, i.e., $\alpha = 1$ and Steiner vertices are allowed, also known as *distance preserving minors*. They showed that general graphs admit distance preserving minors with $O(|K|^4)$ extra non-terminals. A natural question to ask is what is the trade-off between the quality and the number of non-terminals? We make progress on this question from both lower and upper bound perspectives. Specifically, we start by presenting the following lower bound result.

Theorem 1.2.1. *Let $c > 0$ be a constant. For infinitely many $k \in \mathbb{N}$, there exists a graph with k terminals which does not admit an $(\alpha - \epsilon)$-distance approximating minor with k^γ non-terminals, for all $\epsilon > 0$, where α, γ are given in the table below.*

α	2	2.5	3	10/3	11/3	4	4.2
γ	2	5/4	6/5	10/9	11/10	12/11	21/20

To obtain the above result we introduce a novel black-box reduction technique that converts lower bounds for the SPR problem [58] into super-linear lower-bounds on the number of non-terminals for distance approximating minors with the same quality. At the heart of our graph constructions are variants of *Steiner Systems* [258], which are useful concepts studied in combinatorial design. We believe that this connection might be of independent interest.

From the upper bound perspective, we ask the question of whether one can construct $(1+\epsilon)$-distance approximating minors with less than $O(|K|^4)$ non-terminals. For planar graphs, we show this can be actually achieved.

Theorem 1.2.2. *Given a weighted planar graph $G = (V, E, \mathbf{w})$, and a set of terminals K, there exists an algorithm that computes an $(1 + \epsilon)$-distance approximating minor $H = (V', E', \mathbf{w}')$ of G with $|V'| = O(|K|^2\epsilon^{-2}\log^2|K|)$ non-terminals.*

Key to the above result is the notion of *terminal path cover*. At a high level such a cover is a set of shortest paths in the graph whose union (1) contains the terminal set and (2) approximately preserves shortest path distances among terminals. We show that *distance oracles* for planar graphs due to Thorup [245] can be extended to construct terminal path covers for planar graphs. This, combined with the counting argument for branching events in shortest paths of Coppersmith and Elkin [75] proves the claimed guarantees. We remark that our result has been subsequently extended to minor-free graphs by Gupta and DiRenzo [120].

It is an important question whether one can improve the bound on the number of non-terminals in Theorem 1.2.2 while keeping the same quality. In fact, any sub-quadratic bound on the number of non-terminals would imply non-trivial bounds for dynamic planar all-pairs shortest path problem in the offline setting. We refer the reader to Chapter 6 for a detailed treatment on this connection.

1.2.2 Reachability Preserving Minors

Sparsification in directed graphs is usually a much harder task when compared to the undirected counterpart, with many basic graph properties admitting no non-trivial results. In this thesis, we focus on one of most basic graph properties, namely *reachability*, and study it from the vertex sparsification point of view.

Formally, given a directed graph $G = (V, E)$ and a designed subset of terminals K, a *reachability preserving minor* of G is a directed graph $H = (V', E')$ such that (1) $V' \supseteq K$ and V' is small, ideally $|V'| = O(\text{poly}(|K|))$, (2) H is a minor of G and (3) for any pair of vertices $u, v \in K$, there is a directed path from u to v in H if and only if there is a directed path from u to v in G.

We initiate the study of constructing such sparsifiers and provide the first non-trivial guarantees on the problem. Our lower bound shows that, in general, it is not possible to construct reachability preserving minors with a sub-quadratic number of non-terminals.

Theorem 1.2.3. *For infinitely many $k \in \mathbb{N}$ there exists a directed planar graph G with k terminals such that any reachability preserving minor of G must use $\Omega(k^2)$ non-terminals.*

In fact, the graph instance for proving the above lower bound is a *directed acylic grid* with terminals distributed on the boundary of the grid. Our argument essentially proves that all internal, non-boundary vertices of the grid must be retained, if we want to preserve reachability information among terminals. Similar ideas for proving lower-bounds on distance preserving minors for undirected graphs were employed by Krauthgamer, Nguyen, and Zondiner [170].

We complement the lower bound by showing that planar graphs admit reachability sparsifiers with at most $O(|K|^2 \log |K|)$ terminals. For general graphs our bounds are worse only by another $|K|$ factor. Surprisingly, the gaps between the best upper and lower bounds are tighter when compared to distance preserving minors in the undirected setting.

Theorem 1.2.4. *Given a directed graph $G = (V, E)$, and a set of terminals K, there exists an algorithm that computes a rechability preserving minor $H = (V', E')$ of G with $|V'| = O(|K|^3)$ non-terminals. When G is a planar directed graph, the number of non-terminals improves to $|V'| = O(|K|^2 \log |K|)$.*

1.2.3 Structure of Thesis

We start with the fully-dynamic all-pairs effective resistances problem in Chapter 2. We obtain a $(1+\epsilon)$-approximation with $\tilde{O}(\sqrt{n}\epsilon^{-2})$ update and query time on graphs that admit small separators. In the setting where exact effective resistances are required, we show two conditional lower-bounds, one applying to general graphs and the other to graphs that admit small separators, which justify our upper bound that only supports approximate queries. In Chapter 3, we study dynamic algorithms for maintaining vertex spectral sparsifiers with respect to a carefully chosen set

of terminals. We show the applicability of this technique to (1) dynamic Laplacian solvers with $\tilde{O}(n^{11/12}\epsilon^{-5})$ update and query time on unweighted, bounded degree graphs and (2) dynamic $(1+\epsilon)$-approximate all-pairs effective resistances with $\tilde{O}(\min(m^{3/4}, n^{5/6}\epsilon^{-2})\epsilon^{-4})$ update and query time on undirected, unweighted graphs, and $\tilde{O}(n^{5/6}\epsilon^{-6})$ on undirected, weighted graphs.

We then shift our focus to studying tree-based graph approximations in the dynamic setting. In Chapter 4, we develop an algorithm that dynamically maintains a spanning tree with $n^{o(1)}$ average stretch and $O(n^{1/2+o(1)})$ update time on undirected, unweighted graphs. As a by-product of our techniques, we give the best-known running time and size guarantees for the dynamic spanner problem.

In Chapters 5 and 6 we study incremental algorithms for graph partitioning problems. Concretely, in Chapter 5, we show an incremental algorithm for exactly maintaining the value of a global minimum cut in $O(\log^3 n \log\log n)$ update time and $O(1)$ query time. We also design incremental algorithms with small approximation errors that are both time and space efficient. In Chapter 6, we introduce the notion of local sparsifiers and design efficient variants of such sparsifiers for graph properties like distances and cuts. We then show a technique that converts these sparsifiers into incremental algorithms for efficiently maintaining approximate solutions to a range of graph problems including all-pairs minimum cuts and uniform sparsest cut.

The last part of this thesis is devoted to graph sparsification. In Chapter 7, we study distance approximating minors from both a lower and upper bound perspective. For example, we show that for distortion $3-\epsilon$ there are k-terminal graphs for which any distance approximating minor needs to retain at least $\Omega(k^{6/5})$ non-terminals. For planar graphs, we show that there are $(1+\epsilon)$-distance approximating minors with $\tilde{O}(k^2\epsilon^{-2})$ non-terminals. In Chapter 8, we consider reachability preserving minors. We prove that, in general, it is not possible to construct such sparsifiers with a sub-quadratic number of non-terminals and show a matching upper bound on planar graphs, up to a logarithmic factor. We also prove new guarantees for vertex sparsifiers that preserve distance and cuts on planar graphs with terminals lying on the same face.

CHAPTER 2

Dynamic Effective Resistances and Approximate Schur Complement on Separable Graphs

We consider the problem of dynamically maintaining (approximate) all-pairs effective resistances in separable graphs, which are those that admit an n^c-separator theorem for some $c < 1$. We give a fully dynamic algorithm that maintains $(1+\varepsilon)$-approximations of the all-pairs effective resistances of an n-vertex graph G undergoing edge insertions and deletions with $\tilde{O}(\sqrt{n}/\varepsilon^2)$ worst-case update time and $\tilde{O}(\sqrt{n}/\varepsilon^2)$ worst-case query time, if G is guaranteed to be $\sqrt{n}$-separable (i.e., it is taken from a class satisfying a $\sqrt{n}$-separator theorem) and its separator can be computed in $\tilde{O}(n)$ time. Our algorithm is built upon a dynamic algorithm for maintaining *approximate Schur complement* that approximately preserves pairwise effective resistances among a set of terminals for separable graphs, which might be of independent interest.

We complement our result by proving that for any two fixed vertices s and t, no incremental or decremental algorithm can maintain the $s-t$ effective resistance for $\sqrt{n}$-separable graphs with worst-case update time $O(n^{1/2-\delta})$ and query time $O(n^{1-\delta})$ for any $\delta > 0$, unless the Online Matrix Vector Multiplication (OMv) conjecture is false.

We further show that for *general* graphs, no incremental or decremental algorithm can maintain the $s-t$ effective resistance problem with worst-case update time $O(n^{1-\delta})$ and query-time $O(n^{2-\delta})$ for any $\delta > 0$, unless the OMv conjecture is false.

2.1 Introduction

Effective resistances and the closely related electrical flows are basic concepts for resistor networks [86] and were found to be very useful in the design of graph algorithms, e.g., for computing and approximating maximum flow [70, 187, 189], random spanning tree generation [190, 227], multicommodity flow [160], oblivious routing [126], and graph sparsification [83, 235]. They also have found applications in social network analysis, e.g., for measuring the similarity of vertices in social networks [182], in machine learning, e.g., for Gaussian sampling [66] and in chemistry, e.g., for measuring chemical distances [165]. Previous research has studied the problem of how to quickly compute and approximate the effective resistances (or equivalently, *energies* of electrical flows), as such algorithms can be used as a crucial subroutine for other graph algorithms. For example, one can $(1+\varepsilon)$-approximate the $s-t$ effective resistance in $\tilde{O}(m+n\varepsilon^{-2})$ [88] and $\tilde{O}(m\log(1/\varepsilon))$ [74] time, respectively, in any n-vertex m-edge weighted graph, for any two vertices s,t. There are also algorithms that find $(1+\varepsilon)$-approximations to the effective resistance between every pair of vertices in $\tilde{O}(n^2/\varepsilon)$ time [144]. In order to exactly compute the $s-t$ (or single-pair) and all-pairs effective resistance(s), the current fastest algorithms run in times $O(n^\omega)$ (by using the fastest matrix inversion algorithm [57, 141]) and $O(n^{2+\omega})$, respectively, where $\omega < 2.373$ is the matrix multiplication exponent [257]. In planar graphs, the algorithms for exactly computing $s-t$ and all-pairs effective resistance(s) run in times $O(n^{\omega/2})$ (by the nested dissection method for solving linear system in planar graphs [185]) and $O(n^{2+\omega/2})$, respectively.

A natural algorithmic question is how to efficiently maintain the effective resistances *dynamically*, i.e., if the graph undergoes edge insertions and/or deletions, and the goal is to support the update operations and query for the effective resistances as quickly as possible, rather than having to recompute it from scratch each time. Besides the potential applications in the design of other (dynamic) algorithms, it is also of practical interest, e.g., to quickly report the (dis)similarity between any two nodes in a social network in which its members and their relationship are constantly changing. So far our understanding towards this question is very limited: for exact maintenance, the only approach (for single-pair effective resistance) we are aware of is to invoke the dynamic matrix inversion algorithm which gives $O(n^{1.575})$ update time and $O(n^{0.575})$ query time or $O(n^{1.495})$ update time and $O(n^{1.495})$ query time [223]; for $(1+\varepsilon)$-approximate maintenance, we can maintain the spectral sparsifier of size $n\,\mathrm{poly}(\log n,\varepsilon^{-1})$ with $\mathrm{poly}(\log n,\varepsilon^{-1})$ update time [12], while answering each query will cost $\Theta(n\,\mathrm{poly}(\log n,\varepsilon^{-1}))$ time.

In this chapter, we study the problem of dynamically maintaining the (approximate) effective resistances in *separable graphs*, which are those that satisfies an n^c-separator theorem for some $c<1$. Interesting classes of separable graphs include planar graphs, minor free graphs, bounded-genus graphs, almost planar graphs (e.g., road networks) [184], most 3-dimensional meshes [196] and many real-world networks (e.g., phone-call graphs, Web graphs, Internet router graphs) [50]. In the static setting, effective resistances (or electrical flows) in planar/separable graphs

have been utilized by Miller and Peng [194] to obtain the first $\tilde{O}(\frac{m^{6/5}}{\varepsilon^{\Theta(1)}})$ time algorithm for approximate maximum flow in such graphs, and have also been studied by Anari and Oveis Gharan [21] in the analysis of an approximation algorithm for Asymmetric TSP. We now give the necessary definitions to state our results.

Effective Resistances. Let $G = (V, E, \mathbf{w})$ be a undirected weighted graph with $\mathbf{w}(e) > 0$ for any $e \in E$. Let $\mathbf{A}$ denote its weighted adjacency matrix and $\mathbf{D}$ denote the weighted degree diagonal matrix. Let $\mathbf{L} = \mathbf{D} - \mathbf{A}$ denote the *Laplacian* matrix of G. Let $\mathbf{L}^\dagger$ denote the Moore-Penrose pseudo-inverse of the Laplacian of G. Let $\mathbf{1}_u \in \mathbb{R}^V$ denote the indicator vector of vertex u such that $\mathbf{1}_u(v) = 1$ if $v = u$ and 0 otherwise. Let $\boldsymbol{\chi}_{s,t} = \mathbf{1}_s - \mathbf{1}_t$. Given any two vertices $u, v \in V$, the $s - t$ *effective resistance* is defined as $R^G_{\text{eff}}(s, t) := \boldsymbol{\chi}^\top_{s,t} \mathbf{L}^\dagger \boldsymbol{\chi}_{s,t}$.

Separable graphs. Let $\mathcal{C}$ be a class of graphs that is closed under taking subgraphs. We say that $\mathcal{C}$ satisfies a $f(n)$-*separator theorem* if there are constants $\alpha < 1$ and $\beta > 0$ such that every graph in S with n vertices has a cut set with at most $\beta f(n)$ vertices that separates the graph into components with at most αn vertices each [184]. In this chapter we are particularly interested in the class of graphs that satisfies an $n^{1/2}$-separator theorem, which include the class of planar graphs, K_t-minor free graphs and bounded-genus graphs, etc., though our approach can also be generalized to other class of graphs that satisfies a n^c-separator theorem, for some $c < 1$. In the following, we call a graph $f(n)$-*separable* if it is a member of a class that satisfies an $f(n)$-separator theorem.

We would like to quickly maintain the exact or a good approximation of the $s - t$ effective resistances in a $\sqrt{n}$-separable graph that undergoes edge insertions and deletions, for all pairs $s, t \in V$. We call this the *dynamic all-pairs effective resistances problem.* Our goal is to solve this problem with both small update and query times. More precisely, our data structure supports the following operations.

- Insert(u, v, w): Insert the edge (u, v) of weight w to G, provided that the updated graph remains $\sqrt{n}$-separable.
- Delete(u, v): Delete the edge (u, v) from G.
- EffectiveResistance(s, t): Return the exact or approximate value of the effective resistance between s and t in the current graph G.

We remark that our algorithm can be extended to handle operations Increase(u, v, Δ) and Decrease(u, v, Δ) that increases and decreases the weight of any existing edge (u, v) by Δ, respectively, as one can simply delete the edge first and then insert it again with the corresponding new weight. For our lower bound, we will consider the *incremental* (or *decremental*) $s - t$ effective resistance problem, that is, s, t are two vertices fixed at the beginning, and only operations Insert & Decrease (or Delete & Increase) and EffectiveResistance are allowed. The basic idea is that in the incremental (or decremental) setting, the effective resistances

are monotonically decreasing (or increasing) (see e.g., [70]). For any $\varepsilon \in (0,1)$, we say that an algorithm is a $(1+\varepsilon)$-approximation to $R^G_{\text{eff}}(s,t)$ if EffectiveResistance(s,t) returns a positive number k such that $(1-\varepsilon)\cdot R^G_{\text{eff}}(s,t) \leq k \leq (1+\varepsilon)R^G_{\text{eff}}(s,t)$.

2.1.1 Our Results

We give a fully dynamic algorithm for maintaining $(1+\varepsilon)$-approximations of all-pairs and single-pair effective resistance(s) with small update and query times for any $\sqrt{n}$-separable graph, if its separator can be computed fast. Throughout, all the running times of our algorithms are measured in *worst-case* performance. All our algorithms are randomized, and the performance guarantees hold with probability at least $1-n^{-c}$ for some $c \geq 1$. Specifically, we show the following theorem.

Theorem 2.1.1. *Let G denote a dynamic n-vertex graph under edge insertions and deletions. Assume that G is $\sqrt{n}$-separable and its separator can be computed in $s(n)$ time, throughout the updates. There exist fully dynamic algorithms that maintain $(1+\varepsilon)$-approximations of*

- *the all-pairs effective resistances with $\tilde{O}(\frac{\sqrt{n}}{\varepsilon^2}+\frac{s(n)}{\sqrt{n}})$ update time and $\tilde{O}(\frac{\sqrt{n}}{\varepsilon^2})$ query time;*
- *the $s-t$ effective resistance with $\tilde{O}(\frac{\sqrt{n}}{\varepsilon^2}+\frac{s(n)}{\sqrt{n}})$ update time and $O(1)$ query time.*

In particular, if $s(n)=\tilde{O}(n)$, then our update times are $\tilde{O}(\frac{\sqrt{n}}{\varepsilon^2})$.

By using the well known facts that a balanced separator of size $O(\sqrt{n})$ for planar graphs (and bounded-genus graphs) can be computed in $O(n)$ time [184], and for K_t-minor-free graphs (for any fixed integer $t>0$) in $O(n^{1+\delta})$ time, for any constant $\delta>0$ [156], we obtain dynamic algorithms for the effective resistances for planar and minor-free graphs with $\tilde{O}(\sqrt{n}/\varepsilon^2)$ and $\tilde{O}(\sqrt{n}/\varepsilon^2+n^{1/2+\delta})$ update time, respectively.

The performance of our dynamic algorithm in planar graphs almost matches the best-known dynamic algorithm for $(1+\varepsilon)$-approximate all-pairs shortest path in planar graphs with $\tilde{O}(\sqrt{n})$ update and query time [9], though our approaches are different. This is interesting as the shortest path corresponds to flows with controlled ℓ_1 norm while the energy of electrical flows (i.e., effective resistance) corresponds to those with minimum ℓ_2 norm.

In order to design a dynamic algorithm for effective resistances of separable graphs (i.e., to prove Theorem 2.1.1), we give a fully dynamic algorithm that efficiently maintains an *approximate Schur complement* [88, 174, 176] of such graphs (see Section 2.4.1), which might be of independent interest. Approximate Schur complement can be treated as a *vertex sparsifier* that preserves pairwise effective resistances among a set of terminals (see Section 2.3). Therefore, our algorithm is a dynamic algorithm for *vertex effective resistance sparsifiers* with sublinear (in n)

update time for separable graphs. The problem of dynamically maintaining graph *edge sparsifiers* has received attention very recently. For example, Abraham et al. presented fully dynamic algorithms that maintain cut and spectral sparsifiers with poly-logarithmic update times [12]. Formally, we prove the following theorem.

Theorem 2.1.2. *For an n-vertex $\sqrt{n}$-separable graph G whose separator can be computed in $s(n)$ time, and a terminal set $K \subseteq V$ with $|K| \leq O(\sqrt{n})$, there exists a fully dynamic algorithm that maintains a $(1+\delta)$-approximate Schur complement with respect to K' such that $K \subseteq K'$ and $|K'| = O(\sqrt{n})$, while achieving $\tilde{O}(\sqrt{n}/\delta^2 + \frac{s(n)}{\sqrt{n}})$ update time. Furthermore, our algorithm supports terminal additions as long as $|K| \leq O(\sqrt{n})$.*

We complement our algorithm by giving a conditional lower bound for any *incremental* or *decremental* algorithm that maintains *single-pair* effective resistance of a $\sqrt{n}$-separable graph. Our lower bound is established from the *Online Matrix Vector Multiplication* (*OMv*) *conjecture* (see Section 2.2).

Theorem 2.1.3. *No incremental or decremental algorithm can maintain the (exact) $s-t$ effective resistance in $\sqrt{n}$-separable graphs on n vertices with both $O(n^{\frac{1}{2}-\delta})$ worst-case update time and $O(n^{1-\delta})$ worst-case query time for any $\delta > 0$, unless the OMv conjecture is false.*

We note that there are very few conditional lower bounds for dynamic *planar/separable* graphs, as most known reductions are highly non-planar. The only recent result that we are aware of is by Abboud and Dahlgaard [4], who showed that under some popular conjecture, no algorithm for dynamic shortest paths or maximum weight bipartite matching in planar graphs has both updates and queries in amortized $O(n^{1/2-\delta})$ time, for any $\delta > 0$.

We also give a stronger conditional lower bound for the same problem in *general* graphs, which shows that it is hard to maintain effective resistances with both sublinear (in n) update and query times for general graphs, even for the incremental or decremental setting.

Theorem 2.1.4. *No incremental or decremental algorithm can maintain the (exact) $s-t$ effective resistance in general graphs on n vertices with both $O(n^{1-\delta})$ worst-case update time and $O(n^{2-\delta})$ worst-case query time for any $\delta > 0$, unless the OMv conjecture is false.*

We remark that both lower bounds for separable and general graphs hold for any algorithm with sufficiently high accurate approximation ratio, and both lower bounds for incremental algorithms hold even if only edge insertions are allowed (see Section 2.5).

Comparison to [115] In our previous work [115], we gave a fully dynamic algorithm for $(1+\varepsilon)$-approximating all-pairs effective resistances for planar graphs with $\tilde{O}(r/\varepsilon^2)$ update time and $\tilde{O}((r+n/\sqrt{r})/\varepsilon^2)$ query time, for any r larger than some constant. The algorithm can also be generalized to $\sqrt{n}$-separable graphs, and we also provided a conditioned lower bound for any approximation algorithm of the $s-t$ effective resistance in general graphs in the *vertex-update* model. However, besides the apparent improvement of the performance of the dynamic algorithm (i.e., we reduce the best trade off between update time and query time from $\tilde{O}(n^{2/3})$ and $\tilde{O}(n^{2/3})$ to $\tilde{O}(n^{1/2})$ and $\tilde{O}(n^{1/2})$), our current work also improves over and differs from [115] in the following perspectives.

- Our algorithm dynamically maintains the approximate Schur complement of a separable graphs by maintaining a separator tree of such graphs, rather than their r-*divisions* as used in [115]. In fact, we do not believe purely r-divisions based algorithms will achieve the performance as guaranteed by our new algorithm. This is evidenced by previous dynamic algorithms for maintaining reachability in directed planar graphs by Subramanian [240], $(1+\varepsilon)$-approximating to all-pairs shortest paths by Klein and Subramanian [166], exactly maintaining $s-t$ max-flow in planar graphs by Italiano et al. [143], all of which are based on r-divisions and have running times of order $n^{2/3}$ (and some of which have been improved by using other approaches).
- Our current lower bound is much stronger than the previous one: the previous lower bound only holds for general graphs and the *vertex-update* model, where nodes, not edges, are turned on or off, and its proof was based on a simple relation between $s-t$ connectivity and $s-t$ effective resistance $R^G_{\text{eff}}(s,t)$ (i.e., if s,t is connected iff $R^G_{\text{eff}}(s,t)$ is not infinity). In contrast, our new lower bounds hold for separable graphs (and also general graphs) and the edge-update model. The corresponding proofs exploit new reductions from the OMv problem to the 5-length cycle detection and triangle detection problems in separable graphs and general graphs, respectively, which might be of independent interest, and the latter problems are related to the effective resistances (see Section 2.5.1).

2.1.2 Our Techniques

Our dynamic algorithm for maintaining an Approximate Schur complement (ASC) w.r.t. a set of terminals for separable graphs is built upon maintaining a *separator tree* of such graphs and two properties (called *transitivity* and *composability*) of ASCs. Such a tree can be constructed very efficiently by recursively partitioning the subgraphs using separators. Slightly more formally, each node in the tree corresponds to a subgraph of the original graph and contains a subset of vertices as its boundary vertices which in turn are treated as terminals. For each node H, we will maintain an ASC H' of H w.r.t its terminals. We will guarantee throughout all the updates that the ASC of any node can be computed efficiently in a bottom-up fash-

ion, by the above two properties of ASCs. This stems from the fact that we only need to recompute the ASCs of nodes that lie on a path from a *constant* number leaves to the node of interest. Since each such path has length $O(\log n)$ and the recomputation of ASC of one node takes time $\tilde{O}(\sqrt{n})$, the update time will be guaranteed to be $\tilde{O}(\sqrt{n})$. For the detailed implementation, we need to overcome the difficulty that the error in the approximation ratio might accumulate through this recursive computation and an update might require to change the set of boundary vertices of many nodes, thus resulting in a prohibitive running time. We remark that though the idea of using separator tree of planar/separable graphs is standard (e.g., [97]), the main novelty of our algorithm is to use such a tree as the backbone to dynamically maintain the approximate Schur complement.

To obtain our dynamic algorithms for all-pairs effective resistance, we appropriately declare and add new terminals whenever we get a new query, and then run the above dynamic algorithm for ASC with respect to the corresponding terminal set.

To obtain our lower bound, we provide new reductions from the Online Boolean Matrix-Vector Multiplication (OMv) problem to the incremental or decremental single-source effective resistance problem. More specifically, given an OMv instance with vectors $\mathbf{u}, \mathbf{v}$ and a matrix $\mathbf{M}$, we construct a $\sqrt{n}$-separable graph G such that $\mathbf{uMv} = 1$ if and only if there exists a cycle of length 5 incident to some vertex t in G. This 5-length cycle detection problem in turn can be solved by inspecting the diagonal entry corresponding to t of the inverse of a matrix that is defined from G. Furthermore, the diagonal entry of this matrix is inherently related to the effective resistance [198]. By appropriately dynamizing the graph G and using the time bounds for the OMv problem from the conjecture, we get the conditional lower bound for separable graphs.

For general graphs, the lower bound is proved in a similar way, except that the constructed graph is different and we instead use a relation between effective resistance and triangle detection problem. That is, we first reduce the OMv problem to the t-triangle detection problem such that the OMv instance satisfies $\mathbf{uMv} = 1$ if and only if there exists a triangle incident to some vertex t in the constructed G. The latter problem can again be solved by checking the diagonal entry corresponding to t of some matrix, which in turn encodes the effective resistance of between t and a properly specified vertex s.

Other Related Work. Previous work on dynamic algorithms for planar or plane graphs include: shortest paths [9, 143, 166], $s-t$ min-cuts/max-flows [143], reachability in directed graphs [81, 142, 240], (k-edge) connected components [97, 137], the best swap and the minimum spanning forest [97]. There also exist work on dynamic algorithms for $\sqrt{n}$-separable graphs, e.g., on transitive closure and $(1+\varepsilon)$-approximation of all-pairs shortest paths [148].

It is interesting to note that for the (simpler) offline dynamic effective resistance problems, i.e., the sequence of updates and queries are given as an input, Li et

al. [181] recently gave an incremental algorithm with $O(\frac{\text{poly}\log n}{\varepsilon^2})$ amortized update and query time for general graphs.

2.2 Preliminaries

Let $G = (V, E, \mathbf{w})$ be an undirected weighted graph such that $\mathbf{w}(e) > 0$ for any $e \in E$. We fix an arbitrary orientation of edges and treat G as a *resistor network* such that each edge $e \in E$ represents a resistor with *resistance* $\mathbf{r}(e) := 1/\mathbf{w}(e)$. For any vertex pair s, t, the $s - t$ flow is a function $\mathbf{f} : E \to \mathbb{R}^+$ satisfying the *conservation condition*, i.e., for any vertex $v \in V \setminus \{s, t\}$, $\sum_{u:(u,v)\in E} \mathbf{f}(u, v) = \sum_{u:(v,u)\in E} \mathbf{f}(v, u)$. The *energy of an $s - t$ flow* is defined as $\mathcal{E}_G(\mathbf{f}, s, t) := \sum_{e\in E} \mathbf{r}(e)\mathbf{f}(e)^2$. The $s - t$ *electrical flow* $\mathbf{f}^*$ is defined as the $s - t$ flow that minimizes the energy $\mathcal{E}_G(\mathbf{f}, s, t)$ among all $s - t$ flows $\mathbf{f}$ with unit flow value, i.e., $\sum_{v\in V} \mathbf{f}(s, v) = 1$. Let $\mathcal{E}_G(s, t)$ denote the energy of the $s - t$ electrical flow, that is, $\mathcal{E}_G(s, t) := \mathcal{E}_G(\mathbf{f}^*, s, t)$. An electrical flow $\mathbf{f}$ naturally corresponds to a *potential* ϕ in the sense that we can assign each vertex u a potential $\phi(u)$ such that for any $e = (u, v)$, $\mathbf{f}(e) = \frac{\phi(u)-\phi(v)}{\mathbf{r}(e)}$.

It is well known that the $s - t$ effective resistance $R^G_{\text{eff}}(s, t)$ as defined in Section 2.1 satisfies that $R^G_{\text{eff}}(s, t) = \phi(s) - \phi(t)$, which is the potential difference between s, t when we send one unit of the (unique) $s - t$ electrical flow from s to t. Furthermore, it holds that for any s, t, the energy of the $s - t$ electrical flow is equivalent to the $s - t$ effective resistance, that is, $\mathcal{E}_G(s, t) = R^G_{\text{eff}}(s, t)$ (see e.g., [86]). In the following, we will mainly focus on how to dynamically maintain (approximation of) effective resistance $R^G_{\text{eff}}(s, t)$.

Properties of Separable Graphs. Let $G = (V, E)$ be a sparse, $O(\sqrt{n})$-separable graph. For an edge-induced subgraph H of G, any vertex that is incident to vertices not in H is called a *boundary vertex*. We let $\partial(H)$ denote the set of *boundary vertices* belonging to H. All other vertices incident to edges from only H will be called *interior vertices* of H.

A hierarchical decomposition of G is obtained by recursively partitioning the graph using separators into edge-disjoint subgraphs (called regions), where the removal of each separator partitions the subgraph into two two edge-disjoint subgraph. This decomposition is represented by a binary (decomposition) tree $\mathcal{T}(G)$, which we refer to as a *separator tree* of G. For any subgraph H of G, we use $H \in \mathcal{T}(G)$ to denote that H is a node of $\mathcal{T}(G)$ (to avoid confusion with the vertices of G, we refer to the vertices of $\mathcal{T}(G)$ as nodes). The *height* $\eta(H)$ of a node is the number of edges in the longest path between that node and a leaf. In addition, let $S(H)$ denote a balanced separator of the subgraph H. Formally, $\mathcal{T}(G)$ satisfies the following properties:

1. The root node of $\mathcal{T}(G)$ is the graph G.

2. A non-leaf node $H \in \mathcal{T}(G)$ has exactly two children $\mathrm{c}_1(H)$, $\mathrm{c}_2(H)$ and a balanced separator $S(H)$ such that $\mathrm{c}_1(H) \cup \mathrm{c}_2(H) = H$, $V(\mathrm{c}_1(H)) \cap V(\mathrm{c}_2(H)) = S(H)$ and $E(\mathrm{c}_1(H)) \cap E(\mathrm{c}_2(H)) = \emptyset$.
3. For a node $H \in \mathcal{T}(G)$, the set of boundary vertices $\partial(H) \subseteq V(H)$ is defined recursively as follows:
   - If H is the root of $\mathcal{T}(G)$, i.e., $H = G$, then $\partial(G) = S(G)$.
   - Otherwise, $\partial(H) = S(H) \cup (\partial(P) \cap V(H))$, where P is the parent of H in $\mathcal{T}(G)$.
4. For each node $H \in \mathcal{T}(G)$ and its children $\mathrm{c}_1(H)$, $\mathrm{c}_2(H)$, we have $\partial(\mathrm{c}_1(H)) \cup \partial(\mathrm{c}_2(H)) \supseteq \partial(H)$.
5. The number of boundary vertices per node $H \in \mathcal{T}(G)$, i.e., $|\partial(H)|$, is bounded by $O(\sqrt{n})$.
6. There are $O(\sqrt{n})$ leaf subgraphs in $\mathcal{T}(G)$, each having at most $O(\sqrt{n})$ edges.
7. The height of the tree $\mathcal{T}(G)$ is $O(\log n)$, i.e., $\eta(G) = O(\log n)$.
8. Each edge $e \in E$ is contained in a unique leaf subgraph of $\mathcal{T}(G)$.

The lemma below shows that a separator tree can be constructed with an additional $\log n$ factor overhead in the running time for computing a separator. We include its proof here for the sake of completeness.

Lemma 2.2.1. *Let $G = (V, E)$ be a $O(\sqrt{n})$-separable graph whose balanced separator can be computed in $s(n)$ time. There is an algorithm that computes a separator tree $\mathcal{T}(G)$ in $O(s(n) \log n)$ time.*

Proof. For some constant $c \geq 1$, let $S(G)$ be a α-balanced separator of size $c\sqrt{n}$, where $\alpha = 2/3$. First, we let G be the root node of $\mathcal{T}(G)$. Let G_1 and G_2 be the two disjoint components of G obtained after the removal of the vertices in S. We define the children $\mathrm{c}_1(G), \mathrm{c}_2(G)$ of G as follows: $V(\mathrm{c}_i(G)) = V(G_i) \cup S(G)$, $E(\mathrm{c}_i(G)) = E(G_i)$, for $i = 1, 2$, and whenever an edge connects two vertices in $S(G)$, we arbitrarily append it to either $E(\mathrm{c}_1(G))$ or $E(\mathrm{c}_2(G))$. By construction, property (2) in the definition of $\mathcal{T}(G)$ holds. We continue by repeatedly splitting each child $\mathrm{c}_i(G)$ in the same way as we did for G, until there are $O(\sqrt{n})$ components, each of size $O(\sqrt{n})$. The components at this level form the *leaf nodes* of $\mathcal{T}(G)$. Note that the height of $\mathcal{T}(G)$ is bounded by $O(\log n)$ as the size of any child of a node H is at most $2/3$ fraction of the size of H.

We define the boundary vertices for each node in $\mathcal{T}(G)$ according to property (3) in the definition of separator trees. To get the bound on the number of boundary vertices per node $H \in \mathcal{T}(G)$, note that the size of $\partial(H)$ is bounded by

$$\left(c \cdot \sum_{i=0}^{O(\log n)} \sqrt{(2/3)^i} \right) \sqrt{n} = O(\sqrt{n}).$$

Finally, let $t(n)$ be the maximum time required to construct the separator tree of a $O(\sqrt{n})$-separable graph with n vertices. Then, for some suitably chosen n_0,

$$t(n) \leq \begin{cases} s(n) + \max\{t(n_1) + t(n_2)\} & \text{if } n > n_0, \\ 0 & \text{if } n \leq n_0, \end{cases}$$

where the maximum is over n_1, n_2 such that

$$n \leq n_1 + n_2 \leq n + 2c\sqrt{n}, \quad \text{and} \quad \frac{1}{3}n \leq n_i \leq \frac{2}{3}n + c\sqrt{n} \quad \text{for } i = 1, 2.$$

By a similar analysis as the proof of Theorem 1 of [97], we can guarantee that $t(n) \leq O(s(n) \log n)$. □

The OMv Conjecture. Our lower bound will be built upon the following OMv problem and conjecture.

Definition 2.2.2. *In the Online Boolean Matrix-Vector Multiplication (OMv) problem, we are given an integer n and an $n \times n$ Boolean matrix $\mathbf{M}$. Then at each step i for $1 \leq i \leq n$, we are given an n-dimensional column vector $\mathbf{v}_i$, and we should compute $\mathbf{M}\mathbf{v}_i$ and output the resulting vector before we proceed to the next round.*

Conjecture 2.2.3 (OMv conjecture [135])**.** *For any constant $\varepsilon > 0$, there is no $O(n^{3-\varepsilon})$-time algorithm that solves OMv with error probability at most $1/3$.*

We will work on a related problem which is called the **uMv** problem.

Definition 2.2.4. *In the* **uMv** *problem with parameters n_1, n_2, we are given a matrix M of size $n_1 \times n_2$ which can be preprocessed. After preprocessing, a vector pair $\mathbf{u}, \mathbf{v}$ is presented, and our goal is to compute $\mathbf{u}^\top \mathbf{M}\mathbf{v}$.*

Theorem 2.2.5 ([135])**.** *Unless the OMv conjecture 2.2.3 is false, there is no algorithm for the* **uMv** *problem with parameters n_1, n_2 using polynomial preprocessing time and computation time $O(n_1^{1-\delta} n_2 + n_1 n_2^{1-\delta})$ that has an error probability at most $1/3$, for some constant δ.*

Spectral and Resistance Sparsifiers. Below we present two notion of edge sparsifiers. The first requires that the quadratic form of the original and sparsified graph are close. The second requires that all-pairs effective resistances of the corresponding graphs are close.

Definition 2.2.6 (Spectral Sparsifier)**.** *Given a graph $G = (V, E, \mathbf{w})$ and $\varepsilon \in (0, 1)$, we say that a subgraph $H = (V, E_H, \mathbf{w}_H)$ is an $(1 \pm \varepsilon)$-spectral sparsifier of G if*

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

Definition 2.2.7 (Resistance Sparsifier)**.** *Given a graph $G = (V, E, \mathbf{w})$ and $\varepsilon \in (0, 1)$, we say that a subgraph $H = (V, E_H, \mathbf{w}_H)$ is an $(1 \pm \varepsilon)$-resistance sparsifier of G if*

$$\forall u, v \in V,\ (1-\varepsilon)R^G_{\text{eff}}(u, v) \leq R^H_{\text{eff}}(u, v) \leq (1+\varepsilon)R^G_{\text{eff}}(u, v).$$

The following lemma shows that Definition 2.2.6 is equivalent to approximating the pseudoinverse Laplacians. We include its proof here for the sake of completeness.

Lemma 2.2.8. *Assume G is connected. Then the following statements are equivalent:*

1. $\forall \mathbf{x} \in \mathbb{R}^n,\ (1-\varepsilon)\mathbf{x}^\top \mathbf{L}_G \mathbf{x} \leq \mathbf{x}^\top \mathbf{L}_H \mathbf{x} \leq (1+\varepsilon)\mathbf{x}^\top \mathbf{L}_G \mathbf{x}$.
2. $\forall \mathbf{x} \in \mathbb{R}^n,\ \dfrac{1}{(1+\varepsilon)}\mathbf{x}^\top \mathbf{L}_G^\dagger \mathbf{x} \leq \mathbf{x}^\top \mathbf{L}_H^\dagger \mathbf{x} \leq \dfrac{1}{(1-\varepsilon)}\mathbf{x}^\top \mathbf{L}_G^\dagger \mathbf{x}$.

Proof of Lemma 2.2.8. Since $\mathbf{L}_G$ is symmetric we can diagonalize it and write

$$\mathbf{L}_G = \sum_{i=1}^{n-1} \lambda_i^G \mathbf{u}_i \mathbf{u}_i^\top,$$

where $\lambda_1^G \geq \ldots \geq \lambda_{n-1}^G$ are the non-zero sorted eigenvalues of $\mathbf{L}_G$ and $\mathbf{u}_1, \ldots, \mathbf{u}_{n-1}$ are a corresponding set of orthonormal eigenvectors. The *Moore-Penrose Pseudoinverse* of $\mathbf{L}_G$ is then defined as

$$\mathbf{L}_G^\dagger = \sum_{i=1}^{n-1} \frac{1}{\lambda_i^G} \mathbf{u}_i \mathbf{u}_i^\top.$$

We next show that for every $\mathbf{x} \in \mathbb{R}^n$, $(1-\varepsilon)\mathbf{x}^\top \mathbf{L}_G \mathbf{x} \leq \mathbf{x}^\top \mathbf{L}_H \mathbf{x}$ is equivalent to $\mathbf{x}^\top \mathbf{L}_H^\dagger \mathbf{x} \leq \frac{1}{(1-\varepsilon)}\mathbf{x}^\top \mathbf{L}_G^\dagger \mathbf{x}$. The other equivalence can be shown in a symmetric way.

For every $\mathbf{x} \in \mathbb{R}^n$, by definition of $\mathbf{L}_G$ and $\mathbf{L}_H$ we have

$$(1-\varepsilon)\mathbf{x}^\top \mathbf{L}_G \mathbf{x} \leq \mathbf{x}^\top \mathbf{L}_H \mathbf{x} \iff (1-\varepsilon)\sum_{i=1}^{n-1} \lambda_i^G (\mathbf{u}_i^\top \mathbf{x})^2 \leq \sum_{i=1}^{n-1} \lambda_i^H (\mathbf{u}_i^\top \mathbf{x})^2.$$

We next show that

$$\begin{aligned} \forall \mathbf{x} \in \mathbb{R}^n,\ (1-\varepsilon)\sum_{i=1}^{n-1} \lambda_i^G (\mathbf{u}_i^\top \mathbf{x})^2 \leq \sum_{i=1}^{n-1} \lambda_i^H (\mathbf{u}_i^\top \mathbf{x})^2 \\ \iff (1-\varepsilon)\lambda_i^G \leq \lambda_i^H,\ \forall i = 1, \ldots, n-1. \end{aligned} \tag{2.1}$$

Since for every $\mathbf{x} \in \mathbb{R}^n$, $(\mathbf{u}_i^\top \mathbf{x})^2 \geq 0$, $i = 1, \ldots, n-1$, the if-direction of the equivalence in (2.1) follows immediately. For the only-if direction, we proceed by contraposition. To this end, assume that there exists some $i \in \{1, \ldots, n-1\}$ such that $(1-\varepsilon)\lambda_i^G > \lambda_i^H$. Then there exists a vector $\mathbf{x} = \mathbf{u_i} \in \mathbb{R}^n$ such that

$$(1-\varepsilon)\sum_{i=1}^{n-1} \lambda_i^G (\mathbf{u}_i^\top \mathbf{x})^2 = (1-\varepsilon)\lambda_i^G > \lambda_i^H = \sum_{i=1}^{n-1} \lambda_i^H (\mathbf{u}_i^\top \mathbf{x})^2,$$

where the first and last inequality follow from the fact that $\mathbf{u}_i$'s are orthonormal eigenvectors, i.e., $\mathbf{u}_i^\top \mathbf{u}_i = 1$ and $\mathbf{u}_i^\top \mathbf{u}_j = 0, \forall i \neq j$. This gives a contradiction and thus proves the only-if direction. Now, for every $\mathbf{x} \in \mathbb{R}^n$ we have

$$\begin{aligned}
(1-\varepsilon)\mathbf{x}^\top \mathbf{L}_G \mathbf{x} \leq \mathbf{x}^\top \mathbf{L}_H \mathbf{x} &\iff (1-\varepsilon)\lambda_i^G \leq \lambda_i^H,\ \forall i = 1, \ldots, n-1 \\
&\iff \frac{1}{\lambda_i^H} \leq \frac{1}{(1-\varepsilon)} \cdot \frac{1}{\lambda_i^G},\ \forall i = 1, \ldots, n-1 \\
&\iff \sum_{i=1}^{n-1} \frac{1}{\lambda_i^H} (\mathbf{u}_i^\top \mathbf{x})^2 \leq \frac{1}{(1-\varepsilon)} \sum_{i=1}^{n-1} \frac{1}{\lambda_i^G} (\mathbf{u}_i^\top \mathbf{x})^2 \\
&\iff \mathbf{x}^\top \mathbf{L}_H^\dagger \mathbf{x} \leq \frac{1}{(1-\varepsilon)} \mathbf{x}^\top \mathbf{L}_G^\dagger \mathbf{x},
\end{aligned}$$

where the penultimate equivalence can be proven in a similar way to equivalence in (2.1). □

In our algorithm we use the following observations: (1) Since, by definition, the effective resistance between any two nodes u and v is the quadratic form defined by the pseudo-inverse of the Laplacian computed at the vector $\mathbf{1}_s - \mathbf{1}_t$, i.e., $R_{\text{eff}}^G(u,v) = (\mathbf{1}_s - \mathbf{1}_t)^\top \mathbf{L}^\dagger (\mathbf{1}_s - \mathbf{1}_t)$, it follows that the effective resistances between any two nodes in G and H are the same up to a $1/(1 \pm \varepsilon)$ factor. By definitions for resistance and spectral sparsifiers, and Lemma 2.2.8 we have the following fact.

Fact 2.2.9. *Let $\varepsilon \in (0,1)$ and let G be a graph. Then every $(1 \pm \varepsilon)$-spectral sparsifier of G is an $1/(1 \pm \varepsilon)$-resistance sparsifier of G.*

(2) The lemma below suggests that given a graph, by decomposing the graph into several pieces and computing a good sparsifier for each piece, one can obtain a good sparsifier for the original graph which is the union of the sparsifiers for all pieces.

Lemma 2.2.10 ([12], Lemma 4.18)**.** *Let $G = (V, E, \mathbf{w})$ be a weighted graph whose set of edges is partitioned into $E_1, \ldots, E_\ell$. Let H_i be a $(1 \pm \varepsilon)$-spectral sparsifier of $G_i = (V, E_i)$, where $i = 1, \ldots, \ell$. Then $H = \bigcup_{i=1}^{\ell} H_i$ is a $(1 \pm \varepsilon)$-spectral sparsifier of G.*

Schur Complement and Approximate Schur Complement. For a given connected graph $G = (V, E)$ and a set $K \subset V$ of terminals with $1 \leq |K| \leq |V| - 1$, let $F = V \setminus K$ be the set of non-terminal vertices in G. The partition of V into F and K naturally induces the following partition of the Laplacian $\mathbf{L}_G$ into blocks:

$$\mathbf{L}_G = \begin{bmatrix} \mathbf{L}_{[F,F]} & \mathbf{L}_{[F,K]} \\ \mathbf{L}_{[K,F]} & \mathbf{L}_{[K,K]} \end{bmatrix}$$

We remark that since G is connected and F and K are non-empty, one can show that $\mathbf{L}_{[F,F]}$ is invertible. We have the following definition of Schur complement.

Definition 2.2.11 (Schur Complement)**.** *The (unique) Schur complement of a graph Laplacian $\mathbf{L}_G$ with respect to a terminal set K is*

$$\mathbf{SC}(G, K) := \mathbf{L}_{[K,K]} - \mathbf{L}_{[K,F]}\mathbf{L}_{[F,F]}^{-1}\mathbf{L}_{[F,K]}.$$

It is well known that the matrix $\mathbf{SC}(G, K)$ is a Laplacian matrix for some graph G'.

Definition 2.2.12 (Approximate Schur Complement (ASC))**.** *Given a graph $G = (V, E, \mathbf{w})$, $K \subset V$ and its Schur complement $\mathbf{SC}(G, K)$, we say that a graph $H = (K, E_H, \mathbf{w}_H)$ is a $(1 \pm \varepsilon)$-approximate Schur complement (abbr. $(1 \pm \varepsilon)$-ASC) of G with respect to K if*

$$\forall \mathbf{x} \in \mathbb{R}^k,\ (1-\varepsilon)\mathbf{x}^\top \mathbf{SC}(G, K)\mathbf{x} \leq \mathbf{x}^\top \mathbf{L}_H \mathbf{x} \leq (1+\varepsilon)\mathbf{x}^\top \mathbf{SC}(G, K)\mathbf{x}.$$

Moreover, we say that H is an 1-ASC of G with respect to K if $\mathbf{L}_H = \mathbf{SC}(G, K)$.

Note that $(1\pm\varepsilon)$-ASC is a spectral sparsifier of Schur complement. Furthermore, approximate Schur complement can be computed efficiently as guaranteed in the following lemma [88].

Lemma 2.2.13. *Fix $\varepsilon \in (0, 1/2)$ and $\gamma \in (0, 1)$, and let $G = (V, E, \mathbf{w})$ be a graph with $K \subset V$ and $|K| = k$. There is an algorithm $\textsc{ApproxSchur}(G, K, \varepsilon, \delta)$ that computes a $(1 \pm \varepsilon)$-ASC H of G with respect to K such that the following statements hold probability at least $1 - \gamma$:*

1. *The graph H has $O(k\varepsilon^{-2}\log(n/\gamma))$ edges.*
2. *The total running time for computing H is $\tilde{O}(m\log^3(n/\gamma) + n\varepsilon^{-2}\log^4(n/\gamma))$.*

2.3 Useful Properties of Approximate Schur Complement

In this section we show that Approximate Schur complement can be treated as a vertex effective resistance sparsifier, which is a small graph that (approximately) preserves the pairwise effective resistances among terminal vertices of the original graph. Then we show two important properties called *transitivity* and *composability* properties of ASCs, which will be exploited in our dynamic algorithms for ASCs and effective resistances.

ASC as Vertex Resistance Sparsifier. To maintain all-pairs effective resistances efficiently, it will be useful to consider the following notion of *vertex sparsifier* that preserves pairwise effective resistances among a set of terminals.

Definition 2.3.1 (Vertex Resistance Sparsifier (VRS))**.** *Given a graph $G = (V, E, \mathbf{w})$ with $K \subset V$, we say that a graph $H = (K, E_H, \mathbf{w}_H)$ is an $(1 \pm \varepsilon)$-vertex resistance sparsifier (abbr. $(1 \pm \varepsilon)$-VRS) of G with respect to K if*

$$\forall s, t \in K,\ (1-\varepsilon)R_{\text{eff}}^G(s,t) \leq R_{\text{eff}}^H(s,t) \leq (1+\varepsilon)R_{\text{eff}}^G(s,t).$$

We show that ASC can be treated as a vertex resitance sparsifier. For this, we recall the following lemma which shows that the quadratic form of the pseudo-inverse of the Laplacian $\mathbf{L}$ is preserved by taking the quadratic form of the pseudo-inverse of its Schur complement, for demand vectors supported on the terminals.

Lemma 2.3.2 ([194], Lemma 5.1)**.** *Let $\mathbf{b}$ be a demand vector of a graph G whose vertices are partitioned into terminals K, and non-terminals F such that only terminals have non-zero entries in $\mathbf{b}$. Let $\mathbf{b}_K$ be the restriction of $\mathbf{b}$ on the terminals and let $\mathbf{SC}(G,K)$ be the Schur complement of $\mathbf{L}_G$ with respect to K. Then*

$$\mathbf{b}^\top \mathbf{L}_G^\dagger \mathbf{b} = \mathbf{b}_K^\top \mathbf{SC}(G,K)^\dagger \mathbf{b}_K.$$

Using interchangeability between graphs and their Laplacians, we can interpret the above result in terms of graphs as well. The lemma below relates ASCs and vertex resistance sparsifiers. We include its proof here for the sake of completeness.

Lemma 2.3.3. *Let $G = (V, E, \mathbf{w})$ be a graph with $K \subset V$. If H is an $(1 \pm \varepsilon)$-ASC of G with respect to K, then H is an $1/(1 \pm \varepsilon)$-VRS of G with respect to K.*

Proof. Let $k = |K|$. First, note that by Definition 2.2.12 and Lemma 2.2.8 we have

$$\forall \mathbf{x} \in \mathbb{R}^k,\ \frac{1}{(1+\varepsilon)} \mathbf{x}^\top \mathbf{SC}(G,K)^\dagger \mathbf{x} \leq \mathbf{x}^\top {\mathbf{L}_H}^\dagger \mathbf{x} \leq \frac{1}{(1-\varepsilon)} \mathbf{x}^\top \mathbf{SC}(G,K)^\dagger \mathbf{x}.$$

Next, let $(s,t) \in K$ be any terminal pair. Consider the demand vector $\chi_{s,t} \in \mathbb{R}^k$ and extend this vector to $\chi'_{s,t} = \begin{bmatrix} \mathbf{0} & \chi_{s,t} \end{bmatrix}^\top \in \mathbb{R}^n$. By definition of effective resistance and Lemma 2.3.2 we get that

$$\begin{aligned} R^H_{\text{eff}}(s,t) &= \chi_{s,t}^\top {\mathbf{L}_H}^\dagger \chi_{s,t} \leq \frac{1}{(1-\varepsilon)} \chi_{s,t}^\top \mathbf{SC}(G,K)^\dagger \chi_{s,t} \\ &= \frac{1}{(1-\varepsilon)} \chi'^\top_{s,t} \mathbf{L}_G^\dagger \chi'_{s,t} = \frac{1}{(1-\varepsilon)} R^G_{\text{eff}}(s,t). \end{aligned}$$

For the lower-bound on $R^H_{\text{eff}}(s,t)$, using the same reasoning, we get that

$$\begin{aligned} R^H_{\text{eff}}(s,t) &= \chi_{s,t}^\top {\mathbf{L}_H}^\dagger \chi_{s,t} \geq \frac{1}{(1+\varepsilon)} \chi_{s,t}^\top \mathbf{SC}(G,K)^\dagger \chi_{s,t} \\ &= \frac{1}{(1+\varepsilon)} \chi'^\top_{s,t} \mathbf{L}_G^\dagger \chi'_{s,t} = \frac{1}{(1+\varepsilon)} R^G_{\text{eff}}(s,t). \end{aligned}$$

□

Transitivity and Composability of ASCs. In the following, we show a *transitivity* property of ASCs and then show how the ASCs of two neighboring nodes of the separator tree $\mathcal{T}(G)$ can be combined to give the ASC of their parent (called *composability*), which will enable us to compute the ASCs of all nodes of $\mathcal{T}(G)$ in a bottom-up fashion.

Transitivity of ASCs. To show the transitivity property the ASCs, we will use the following lemma which establishes the connection between the Schur complement and the Laplacian of the original graph.

Lemma 2.3.4 ([194], Lemma B.2)**.** *Let $\mathbf{L}_G$ be the Laplacian of G and let $\mathbf{SC}(G, K)$ be its Schur complement. For any $x \in \mathbb{R}^k$ the following holds*

$$\mathbf{x}^\top \mathbf{SC}(G, K)\mathbf{x} = \min_{\mathbf{y}} \begin{bmatrix} \mathbf{y} \\ \mathbf{x} \end{bmatrix}^\top \mathbf{L}_G \begin{bmatrix} \mathbf{y} \\ \mathbf{x} \end{bmatrix}.$$

We are now ready to show the following transitive property of ASCs.

Lemma 2.3.5 (Transitivity of ASCs)**.** *If H' is an $(1 \pm \varepsilon)$-ASC of G with respect to K', and H is an $(1 \pm \varepsilon)$-ASC of H' with respect to K, where $K' \supseteq K$, then H is an $(1 \pm \varepsilon)^2$-ASC of G with respect to K.*

Proof. Let $k = |K|$ and $k' = |K'|$. By the assumption of the lemma, the following inequalities hold:

$$\forall \mathbf{x} \in \mathbb{R}^{k'},\ (1-\varepsilon)\mathbf{x}^\top \mathbf{SC}(G, K')\mathbf{x} \leq \mathbf{x}^\top \mathbf{L}_{H'}\mathbf{x} \leq (1+\varepsilon)\mathbf{x}^\top \mathbf{SC}(G, K')\mathbf{x},$$

and

$$\forall \mathbf{x} \in \mathbb{R}^{k},\ (1-\varepsilon)\mathbf{x}^\top \mathbf{SC}(H', K)\mathbf{x} \leq \mathbf{x}^\top \mathbf{L}_{H}\mathbf{x} \leq (1+\varepsilon)\mathbf{x}^\top \mathbf{SC}(H', K)\mathbf{x}.$$

We need to show that

$$\forall \mathbf{x} \in \mathbb{R}^{k},\ (1-\varepsilon)^2\mathbf{x}^\top \mathbf{SC}(G, K)\mathbf{x} \leq \mathbf{x}^\top \mathbf{L}_{H}\mathbf{x} \leq (1+\varepsilon)^2\mathbf{x}^\top \mathbf{SC}(G, K)\mathbf{x}.$$

We first show the upper bound on $\mathbf{x}^\top \mathbf{L}_H \mathbf{x}$. Note that since $K' \supseteq K$, using Gaussian elimination, $\mathbf{SC}(G, K)$ can be constructed by first constructing $\mathbf{SC}(G, K')$ from G and then constructing $\mathbf{SC}(G, K)$ from $\mathbf{SC}(G, K')$ using Gaussian elimination. Thus $\mathbf{SC}(G, K)$ is the Schur complement of $\mathbf{SC}(G, K')$ with respect to K. For any $\mathbf{x} \in \mathbb{R}^k$, let $\mathbf{y}$ be the vector that attains the minimum value in Lemma 2.3.4 for $\mathbf{SC}(G, K')$. If we define $\mathbf{x}' = \begin{bmatrix} \mathbf{y} & \mathbf{x} \end{bmatrix}^\top \in \mathbb{R}^{k'}$, then we get

$$\begin{aligned} \mathbf{x}^\top \mathbf{L}_H \mathbf{x} &\leq (1+\varepsilon)\mathbf{x}^\top \mathbf{SC}(H', K)\mathbf{x} \leq (1+\varepsilon)\mathbf{x'}^\top \mathbf{L}_{H'}\mathbf{x'} \\ &\leq (1+\varepsilon)^2 \mathbf{x'}^\top \mathbf{SC}(G, K')\mathbf{x'} = (1+\varepsilon)^2 \mathbf{x}^\top \mathbf{SC}(G, K)\mathbf{x}. \end{aligned}$$

We now give the lower bound on $\mathbf{x}^\top \mathbf{L}_H \mathbf{x}$. Recall that $\mathbf{SC}(H', K)$ is the Schur complement of $\mathbf{L}_{H'}$ with respect to K. For any vertex $\mathbf{x} \in \mathbb{R}^k$, let $\mathbf{y}$ be the vector given by Lemma 2.3.4 for $\mathbf{L}_{H'}$. If we define $\mathbf{x}'' = \begin{bmatrix} \mathbf{y} & \mathbf{x} \end{bmatrix}^\top \in \mathbb{R}^{k'}$, then we get

$$\begin{aligned} \mathbf{x}^\top \mathbf{L}_H \mathbf{x} &\geq (1-\varepsilon)\mathbf{x}^\top \mathbf{SC}(H', K)\mathbf{x} = (1-\varepsilon)\mathbf{x''}^\top \mathbf{L}_{H'}\mathbf{x''} \\ &\geq (1-\varepsilon)^2 \mathbf{x''}^\top \mathbf{SC}(G, K')\mathbf{x''} \geq (1-\varepsilon)^2 \mathbf{x}^\top \mathbf{SC}(G, K)\mathbf{x}. \end{aligned}$$

□

Composability of ASCs. To show the composability of ASCs, we first review an equivalent way of defining Schur complements. The main idea is to view $\mathbf{SC}(G, K)$ as a multi-graph where each multi-edge corresponds to a walk in G that starts and ends at K, but has all intermediate vertices in $V \setminus K$. We call such a walk a *terminal-free* walk that starts and ends in K. Formally, a terminal-free walk

$$u_0, \ldots, u_\ell$$

of length ℓ, with $u_0, u_\ell \in K$ and $u_i \in V \setminus K$, for $i = 1, \ldots, \ell$ corresponds to a multi-edge between u_0 and u_ℓ in $\mathbf{SC}(G, K)$ with weight given by

$$\mathbf{w}^{\mathbf{SC}(G,K)}_{u_0,\ldots,u_\ell} = \frac{\prod_{0\le i\le \ell} \mathbf{w}(u_i, u_i + 1)}{\prod_{0<i<\ell} \mathbf{d}(u_i)}, \tag{2.2}$$

where $\mathbf{d}(u) = \sum_{v:(u,v)\in E} \mathbf{w}(u, v)$ denotes the weighted degree of a vertex u.

This connection is formally proven in the lemma below.

Lemma 2.3.6 ([89], Lemma 5.4)**.** *Given a graph G and a partition of its vertices into K and $V \setminus K$, the graph G^K obtained by forming an union over all multi-edges corresponding to terminal-free walks that start and end in K, with weights given by Equation (2.2) is exactly $\mathbf{SC}(G, K)$.*

We next show that if a graph can be viewed as a combination of two graphs along some subset of shared terminals, combining the respective sparsifiers of these two graphs in the same way gives a sparsifer for the original graph.

Formally, Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be edge-disjoint graphs with terminals K_1 and K_2, respectively. Furthermore, assume that all vertices in the intersection of V_1 and V_2, if exist, are terminals in both graphs. That is, $(V_1 \cap V_2) \subset K_i$, for $i = \{1, 2\}$. The *merge* of G_1 and G_2 is the graph $G = (V_1 \cup V_2, E_1 \cup E_2)$ with terminals $K_1 \cup K_2$ formed by identifying the terminals in S. We denote this operation by $G := G_1 \oplus G_2$.

Lemma 2.3.7 (Composability of Schur complement)**.** *Let $G := G_1 \oplus G_2$. If H_1 is an 1-ASC of G_1 with respect to K_1, and H_2 is an 1-ASC of G_2 with respect to K_2, then $H := H_1 \oplus H_2$ is an 1-ASC of G with respect to K.*

Proof. Note that $H_i = \mathbf{SC}(G_i, K_i)$, for $i = \{1, 2\}$, and recall that the G_1 and G_2 share the terminals in some non-empty subset S, i.e., $S \subset K_i$, for $i = \{1, 2\}$. To prove the lemma, we need to show that

$$\mathbf{SC}(G_1, K_1) \oplus \mathbf{SC}(G_2, K_2) = \mathbf{SC}(G, K).$$

We do so by making use of Lemma 2.3.6. More specifically, we argue that every multi-edge (along with its corresponding weight) in $\mathbf{SC}(G, K)$ is contained either in $\mathbf{SC}(G_1, K_1)$ or $\mathbf{SC}(G_2, K_2)$. We distinguish the following cases.

(1) For any two terminals t and t' in $K_1 \setminus S$, we have that $\mathbf{SC}(G_1, K_1)$ contains all the multi-edges between t and t' in $\mathbf{SC}(G, K)$. This is because G_1 and G_2 are

edge-disjoint, and there is no terminal-free walk between t and t' in G that does not use a terminal in S. The same reasoning can be applied to terminal pairs in $K_2 \setminus S$.

(2) For any two terminals s and t in $S \times K$, we have that the corresponding multi-edges in $\mathbf{SC}(G, K)$, are either contained in $\mathbf{SC}(G_1, K_1)$ or $\mathbf{SC}(G_2, K_2)$. If $t \in K_1 \setminus S$ or $t \in K_2 \setminus S$, then the same reasoning as in case (1) applies. However, if $t \in S$, then $S(G_1, K_1)$ contains all the multi-edges that correspond to terminal-free walks between s and t that use the edges in G_1, and $S(G_2, K_2)$ contains all the multi-edges that correspond to terminal-free walks between s and t that use the edges in G_2.

(3) For any two terminals t and t' in $(K_1 \setminus S) \times (K_2 \setminus S)$, there is no terminal-free walk between t and t' in G that does not use a terminal in S, since S is a separator of G. Thus there are no multi-edges between t and t' in $\mathbf{SC}(G, K)$, so the merge $\mathbf{SC}(G_1, K_1) \oplus \mathbf{SC}(G_2, K_2)$ correctly does not add such edges. □

Lemma 2.3.8 (Composition of ASCs)**.** *Let $G := G_1 \oplus G_2$. If H_1' is an $(1 \pm \varepsilon)$-ASC of G_1 with respect to K_1, and H_2' is an $(1 \pm \varepsilon)$-ASC of G_2 with respect to K_2, then $H' := H_1' \oplus H_2'$ is an $(1 \pm \varepsilon)$-ASC of G with respect to K.*

Proof. First, let H_1 be an 1-ASC of G_1 with respect to K_1, and H_2 be an 1-ASC of G_2 with respect to K_2. By Lemma 2.3.7, $H := H_1 \oplus H_2$ is an 1-ASC of G with respect to K, i.e., $\mathbf{L}_H = \mathbf{SC}(G, K)$. Now note that we can treat H_i and H_i', for $i = \{1, 2\}$ as graphs defined on the same vertex set $V(H)$, by adding appropriate isolated vertices. By assumption, each H_i' is an $(1 \pm \varepsilon)$-spectral sparsifier of H_i and thus, applying the Decomposition Lemma 2.2.10 gives that $H' := H_1' \oplus H_2'$ is an $(1 \pm \varepsilon)$-spectral sparsifier of H, or equivalently, H' is an $(1 \pm \varepsilon)$-ASC of G. □

2.4 Dynamic Effective Resistances on Separable Graphs

In this section, we first present our fully dynamic algorithm for maintaining a $(1 \pm \delta)$-approximate Schur complement (i.e., prove Theorem 2.1.2) and then use it give a dynamic algorithm for $(1 + \varepsilon)$-approximating all-pairs effective resistances in separable graphs and prove Theorem 2.1.1. For simplicity, we assume that the separator of G can be computed in $\tilde{O}(n)$ time.

2.4.1 Dynamic Approximate Schur Complement

Let $\delta \in (0, 1)$. Let $K \subset V$ be a set of terminals with $|K| \leq O(\sqrt{n})$. We give a data-structure for maintaining a $(1 \pm \delta)$-ASC of a $O(\sqrt{n})$-separable graph G with respect to a set K' of $O(\sqrt{n})$ vertices (which contains the terminal set K) that supports Insert and Delete operations as defined before. In addition, it supports the following operation:

- AddTerminal(u): Add the vertex u to the terminal set K, as long as $|K| \leq O(\sqrt{n})$.

Algorithm 2.1: ApproxSchurNode$(H, \partial(H), \delta')$

```
Set γ ← 1/n^3
if H is a leaf then
    Set H' ← ApproxSchur(H, ∂(H), δ', γ)
if H is a non-leaf then
    Let c_1(H), c_2(H) be the children of H
    Let c_i(H)' be the ASC of c_i(H), for i = 1, 2
    Set R ← c_1(H)' ⊕_φ c_2(H)' and E(R) ← E(R) ∪ X(H)
    Set H' ← ApproxSchur(R, ∂(H), δ', γ)
return H'
```

Data Structure. Throughout we compute and maintain a balanced separator $S(G)$ of G that contains K and satisfies that $|S(G)| \leq O(\sqrt{n})$. We let $K' = S(G)$ and we will maintain a $(1 \pm \delta)$-ASC of G w.r.t. K'. By definition of boundary vertices, $K' = \partial(G)$. Let $\delta' = \frac{\delta}{c \log n + 1}$ for some constant c. In our dynamic algorithm, we will maintain a separator tree $\mathcal{T}(G)$ (see Section 2.2) such that for each node $H \in \mathcal{T}(G)$, we maintain its separator $S(H)$ and a set $\mathrm{X}(H)$ of edges of H, which is initially empty, and an ASC H' of H w.r.t. $\partial(H)$. Throughout the updates, the set $\mathrm{X}(H)$ will denote the subset of edges which are only contained in H while contained in neither of its children. Let $\mathcal{D}(G, \delta)$ denote such a data-structure. We recompute $\mathcal{D}(G, \delta)$ every $\Theta(\sqrt{n})$ operations using the initialization below.

Initialization. We show how to efficiently compute the ASC H' for each node H from $\mathcal{T}(G)$. We do this in a bottom-up fashion by first calling Algorithm 1 on each leaf node and then on the non-leaf nodes, where ApproxSchur is the procedure from Lemma 2.2.13.

In what follows, whenever we compute an approximate Schur complement, we assume that procedure ApproxSchur from Lemma 2.2.13 is invoked on the corresponding subgraph and its boundary vertices, with error δ' and error probability $\gamma = 1/n^3$. In the following, we will assume that all the calls to the ApproxSchur are correct.

The following lemma shows that after invoking Algorithm 1 in a bottom-up fashion, we have computed the ASC for every node in $\mathcal{T}(G)$.

Lemma 2.4.1. *Let $H \in \mathcal{T}(G)$ be a node of height $\eta(H) \geq 0$ and $\mathrm{X}(H) = \emptyset$. Then $H' =$ ApproxSchur Node$(H, \partial(H), \varepsilon)$ is an $(1 \pm \delta')^{\eta(H)+1}$-ASC of H with respect to $\partial(H)$.*

Proof. We proceed by induction on $\eta(H)$. For the base case, i.e., $\eta(H) = 0$, H is a leaf node. By Lemma 2.2.13 and Algorithm 1, H' is indeed a $(1 \pm \delta')$-ASC of H with respect to $\partial(H)$.

Let H be a non-leaf node, i.e. $\eta(H) > 0$. Let $\mathrm{c}_1(H), \mathrm{c}_2(H)$ and $\mathrm{c}'_1(H), \mathrm{c}'_2(H)$ be defined as in Algorithm 1. By properties (2), (3) and (4) of $\mathcal{T}(G)$ and the fact that

$X(H) = \emptyset$, we have $H = \mathrm{c}_1(H) \oplus \mathrm{c}_2(H)$. By induction hypothesis, it follows that $\mathrm{c}_i(H)'$ is an $(1 \pm \delta')^{\eta(\mathrm{c}_i(H))+1}$-ASC of $\mathrm{c}_i(H)$, for $i = 1, 2$. Using Lemma 2.3.8 and since $\eta(\mathrm{c}_i(H)) + 1 = \eta(H)$, for $i = 1, 2$, we get that $R := \mathrm{c}_1(H)' \oplus \mathrm{c}_2(H)'$ is an $(1 \pm \delta')^{\eta(H)}$-ASC of H with respect to $V(R) := \partial(\mathrm{c}_1(H)) \cup \partial(\mathrm{c}_2(H))$. Now, since $V(R) \supseteq \partial(H)$ by property (4) of $\mathcal{T}(G)$ and by Lemma 2.2.13, it follows that H' is an $(1 \pm \delta')$-ASC of R with respect to $\partial(H)$. Finally, applying Lemma 2.3.5 on R and H' we get that H' is an $(1 \pm \delta')^{\eta(H)+1}$-ASC of H. □

Next we analyze the running time of the initialization and recomputation procedure. The lemma below shows that the ASC of any node in $\mathcal{T}(G)$ can be computed in $\tilde{O}(\sqrt{n}/\delta^2)$.

Lemma 2.4.2. *Let $H \in \mathcal{T}(G)$ and assume that $|X(H)| \leq O(\sqrt{n})$. Then we can compute an ASC $H' =$ ApproxSchurNode$(H, \partial(H), \varepsilon)$ of H in $\tilde{O}(\sqrt{n}/\delta^2)$ time.*

Proof. We distinguish two cases. First, if H is a leaf node, then by property (5) of $\mathcal{T}(G)$, we have that $|E(H)| \leq O(\sqrt{n})$. The latter along with Lemma 2.2.13 (2) imply the time to compute H' is $\tilde{O}(\sqrt{n}/\delta'^2)$. Second, if H is a non-leaf node, then by Lemma 2.2.13 (1) we know that $|E(\mathrm{c}_i(H)')| \leq \tilde{O}(\sqrt{n}/\delta'^2)$, for $i = 1, 2$. Since by assumption $|\mathrm{X}(H)| \leq O(\sqrt{n})$, we get that $|R \cup \mathrm{X}(H)| \leq \tilde{O}(\sqrt{n}/\delta'^2)$. Thus, the time to compute H' on top of $R \cup \mathrm{X}(H)$ is bounded by $\tilde{O}(\sqrt{n}/\delta'^2) = \tilde{O}(\sqrt{n}/\delta^2)$ (again by Lemma 2.2.13 (2) and the choice of δ'). □

We now analyze the running time for initializing our data-structure. Let $T_{\mathcal{D}(G)}$ denote the time required to compute $\mathcal{D}(G)$.

Lemma 2.4.3. *The time $T_{\mathcal{D}(G)}$ required to compute $\mathcal{D}(G)$ is $\tilde{O}(n/\delta^2)$.*

Proof. By Lemma 2.2.1 recall that we can construct $\mathcal{T}(G)$ in $\tilde{O}(n)$ time. Note that by construction of the separator tree, the number of non-leaf nodes is bounded by the number of leaf nodes. Since there there are $O(\sqrt{n})$ leaf nodes, the total number of nodes in $\mathcal{T}(G)$ is $O(\sqrt{n})$. By Lemma 2.4.2 we get that the time needed to compute an ASC H' for every node $H \in \mathcal{T}(G)$ is $\tilde{O}(\sqrt{n}/\delta^2)$. Combining the above bounds gives that $T_{\mathcal{D}(G)}$ is $\tilde{O}(n/\delta^2)$. □

Since $\delta' = \frac{\delta}{c \log n + 1}$ and $\eta(G) = O(\log n)$, the graph G' is a $(1 \pm \delta)$-ASC of G w.r.t. $\partial(G)$.

Handling Edge Insertions. We now describe the Insert operation. Let us consider the insertion of an edge $e = (u, v)$ of weight w. We maintain a stack Q, which is initially set to empty. We then update the root node by adding (u, v) with weight w to G, and push G onto Q. During the traversal of $\mathcal{T}(G)$, our procedure maintains two pointers that point to the current node H (initially set to G) and a node N (if any exists) that represents the node for which u and v belong to different children of N, respectively. As long as we have not found such a node N, and the current node H is not a leaf, we proceed as follows.

Algorithm 2.2: UPDATEAPPROXSCHUR(STACKQ)

```
while Q ≠ ∅ do
    Set H ← Q.PULL()
    Set H' ← APPROXSCHURNODE(H, ∂(H), ε)
```

Algorithm 2.3: INSERT(u, v, w)

```
Let Q be an initially empty stack.
Set E(G) ← E(G) ∪ {(u, v)}, Q.PUSH(G), H ← G and N ← NIL
while N = NIL and H is a non-leaf do
    if there exists a child of H that contains both u and v then
        Let c(H) denote any such a child
        Set E(c(H)) ← E(c(H)) ∪ {(u, v)}
        Set H ← c(H)
        Q.PUSH(H)
    else
        Set N ← H
        Set X(N) ← X(N) ∪ {(u, v)}
        ADDBOUNDARY(u, N), ADDBOUNDARY(v, N)
// Update the ASCs of the nodes in Q
UPDATEAPPROXSCHUR(Q)
```

We examine the child of H that contains both u and v (if there is more than one, then we just pick one of them). If u and v belong to the same child, say $\mathrm{c}(H)$, then we add this edge to $\mathrm{c}(H)$ and update the current node H to $\mathrm{c}(H)$. We then push H onto Q. If, however, u and v belong to different children, then we set N to be the current node H and add the edge (u, v) to $\mathrm{X}(N)$, since u and v cannot appear together in the nodes of the lower levels. At this point, this forces u and v to become boundary vertices in N and all other nodes descending from N that contain either u or v. We handle this by making use of the ADDBOUNDARY() procedure, depicted in Algorithm 2.4. Finally, we recompute the ASCs of the affected nodes in a bottom-up fashion using the stack Q (as shown in Algorithm 2.2). This procedure is summarized in Algorithm 2.3. We remark that for simplicity, we let Q.PUSH(H) denote the event of pushing the pointer to H to the stack Q, for any node H.

After the pre-processing step and after each insertion/deletion of an edge, our augmented separator tree $\mathcal{T}(G)$ satisfies the following invariant.

Invariant 2.4.4. *For every edge e in the current graph G, exactly one of the following two holds:*

- *there is a leaf node $H \in \mathcal{T}(G)$ such that $e \in E(H)$,*
- *there is an internal node $H \in \mathcal{T}(G)$ such that $e \in X(H)$.*

The following lemma guarantees that the updated graph G' (i.e., the sparsifier of the root node G) is good approximation to the Schur complement of G with respect to the boundary, after the execution of INSERT(u, v) in Algorithm 2.3.

Lemma 2.4.5. *Let G' be the updated sparsifier of the root node G, after the insertion of edge (u, v). Then G' is an $(1 \pm \delta)$-ASC of G with respected to $\partial(G)$.*

Proof. We proceed inductively as in the proof of Lemma 2.4.1 and show that for any node H, the corresponding sparsifier H' is an $(1 \pm \delta')^{\eta(H)+1}$-ASC of H with respect to $\partial(H)$. Since the base case remains the same, let us consider a non-leaf node H. If $\mathrm{X}(H) = \emptyset$, then the correctness follows from the inductive step of Lemma 2.4.1. However, $\mathrm{X}(H) \neq \emptyset$ implies that $H \neq \mathrm{c}_1(H) \oplus \mathrm{c}_2(H)$. This is because H is the last node for the edges of $\mathrm{X}(H)$ whose endpoints were contained in the same node in $\mathcal{T}(G)$. Recall that the endpoints of all the edges in $\mathrm{X}(H)$ were declared boundary vertices for H and all descendants containing them. Thus we have that

$$H = (\mathrm{c}_1(H) \oplus \mathrm{c}_2(H)) \cup \mathrm{X}(H).$$

By induction hypothesis, it follows that $c_i(H)'$ is an $(1 \pm \delta')^{\eta(\mathrm{c}_i(H))+1}$-ASC of $c_i(H)$, for $i = 1, 2$. Using Lemma 2.3.8 and since $\eta(\mathrm{c}_i(H)) + 1 = \eta(H)$, for $i = 1, 2$, we get that $R := \mathrm{c}_1(H)' \oplus_\phi \mathrm{c}_2(H)'$ is an $(1 \pm \delta')^{\eta(H)}$-ASC of $H \setminus \mathrm{X}(H)$ with respect to $V(R) := \partial(\mathrm{c}_1(H)) \cup \partial(\mathrm{c}_2(H))$. First, since $V(R) \supseteq V(\mathrm{X}(H))$ by construction, Lemma 2.3.8 implies that $R' := R \cup \mathrm{X}(H)$ is an $(1 \pm \delta')^{\eta(H)}$-ASC of $(H \setminus \mathrm{X}(H)) \cup \mathrm{X}(H) = H$ with respect to $V(R)$. Second, since $V(R) \supseteq \partial(H)$ by property (4) of $\mathcal{T}(G)$ and by Lemma 2.2.13, it follows that H' is an $(1 \pm \delta')$-ASC of R' with respect to $\partial(H)$. Finally, applying Lemma 2.3.5 on R' and H' we get that H' is an $(1 \pm \delta')^{\eta(H)+1}$-ASC of H. The statement of the lemma then follows from the facts that $\delta' = \frac{\delta}{c \log n + 1}$ and $\eta(G) = O(\log n)$. □

For the running time of INSERT(u, v, w), we distinguished two cases.

First, suppose that the insertion of the edge (u, v) does not trigger a re-computation of the data-structure. Note that the stack Q (in Algorithm 2.3) contains all nodes in the path starting from the root node G, and then repeatedly choosing *exactly one* child of the current node that contains both u and v, until the node N is reached. Since the height of $\mathcal{T}(G)$ is $O(\log n)$, it follows that $|Q| \leq O(\log n)$. Additionally, by Lemma 2.4.2, the time to re-compute an ASC of any node is bounded by $\tilde{O}(\sqrt{n}/\delta^2)$. Thus we get that the time needed to update the ASCs of the nodes in Q is $\tilde{O}(\sqrt{n}/\delta^2)$. As we will shortly argue, the running time of ADDBOUNDARY() is also bounded by $\tilde{O}(\sqrt{n}/\delta^2)$. Combining the above, we get that the running time of INSERT(u, v) is $\tilde{O}(\sqrt{n}/\delta^2)$.

Second, suppose that the edge (u, v) triggers a re-computation of the data-structure. Then by Lemma 2.4.3, we recompute $\mathcal{D}(G, \delta)$ in $\tilde{O}(n/\delta^2)$ time. Since we recompute that data-structure every $\Theta(\sqrt{n})$ insertions, the amortized update time per insertion is $\tilde{O}(\sqrt{n}/\delta^2)$. The above bounds combined give that the amortized time per edge insertion is bounded by $\tilde{O}(\sqrt{n}/\delta^2)$. This bound can be made

Algorithm 2.4: AddBoundary(u, v, w)

```
Let Q be an initially empty stack. while N = nil do
    if u ∉ ∂(H) then
        Set ∂(H) ← ∂(H) ∪ {u}
        Q.Push(H)
        if H is a non-leaf then
            Let c(H) be the unique child that contains u
            Set H ← c(H)
    if H is a leaf then
        Set H ← nil
// Update the ASCs of the nodes in Q
UpdateApproxSchur(Q).
```

worst-case by keeping two copies of the data structure and performing periodical rebuilds.

Handling Terminal Additions to the Boundary. We now describe the AddTerminal(u) operation. It is implemented by simply invoking AddBoundary(u, G), where G is the root of $\mathcal{T}(G)$. For the procedure AddBoundary(u, H), we maintain a stack Q, which is initially set to empty. As long as the current H is a node in $\mathcal{T}(G)$, we first check whether $u \in \partial(H)$. If this is the case, then we simply do nothing as the ASC H' of H with respect to $\partial(H)$ contains u. Otherwise, we add u to $\partial(H)$, and push the node H to Q. Next, if H is not a leaf-node, let $\mathrm{c}(H)$ be the *unique* child that contains u. We then set $\mathrm{c}(H)$ to be our current node H and perform the same steps as above, until we reach some leaf-node, in which case we set H to nil. Finally, we recompute the ASCs of the affected nodes in a bottom-up fashion using the stack Q. This procedure is summarized in Algorithm 2.4.

The correctness of this procedure can be shown similarly to the correctness of Insert(). For the running time, the crucial observation is that if $u \notin \partial(H)$, for some non-leaf node H, then by property (2) of $\mathcal{T}(G)$, it follows that u is assigned to an *unique* child of H. Thus, in the worst-case, the stack Q contains all the nodes in the path between H and some leaf-node. Note that $|Q| = O(\log n)$ and by Lemma 2.4.2, time to re-compute an ASC of any node is $\tilde{O}(\sqrt{n}/\delta^2)$. Combining the above, we get that the running time of AddBoundary(u, H) is $\tilde{O}(\sqrt{n}/\delta^2)$.

Handling Edge Deletions. We now describe the Delete operation. Let us consider the deletion of an edge $e = (x, y)$. Our procedure is symmetric to the Insert() operation. We maintain a stack Q, which is initially set to empty. We then update the root node by deleting (u, v) from G, and pushing G onto Q. During the traversal of $\mathcal{T}(G)$, our procedure maintains the current node H (initially set to G) and determines the node N that represent the lowest-level node in $\mathcal{T}(G)$ that contains

Algorithm 2.5: DELETE(u, v)

```
Let Q be an initially empty stack.
Set E(G) ← E(G) \ {(u, v)}, Q.PUSH(G), H ← G and N ← NIL.
while N = NIL do
    if If there exists a (unique) child c(H) of H that contains (u, v) then
        E(c(H)) ← E(c(H)) \ {(u, v)}.
        Set H ← c(H).
        Q.PUSH(H).
    else
        Set N ← H.
    if N is a non-leaf then
        X(N) ← X(N) \ {(u, v)}.
// Update the ASCs of the nodes in Q
UPDATEAPPROXSCHUR(Q).
```

the edge (u, v). Note that N is not necessarily a leaf-node. As long as we have not found such a node we proceed as follows.

We examine the *unique* child of H that contains the edge (u, v) (by property (2) of $\mathcal{T}(G)$). If there exists such a child $c(H)$, then we delete (u, v) from $c(H)$ and update the current node H to $c(H)$. We then push H to Q. If, however, such a child does not exist, then we set N to be the current node H. Next, if N is a non-leaf node, we remove the edge (u, v) from $\mathrm{X}(N)$. Finally, we recompute the ASCs of the affected nodes in a bottom-up fashion using the stack Q. This procedure is summarized in Algorithm 2.5.

Similarly to the INSERT() operation, we can show that the worst-case running time of DELETE(u, v) operation is $\tilde{O}(\sqrt{n}/\delta^2)$.

Finally, recall that we set $\gamma = 1/n^3$ as the error probability of APPROXSCHUR from Lemma 2.2.13. This will guarantee that throughout all updates, our algorithm succeeds with probability at least $1 - O(n) \cdot \frac{1}{n^3} \geq 1 - O(\frac{1}{n^2})$ as the total number of nodes in $\mathcal{T}(G)$ is $O(\sqrt{n})$, each update involves recomputation of the ASCs of $O(\log n)$ nodes and our algorithm recomputes the data structure every $\Theta(\sqrt{n})$ operations.

Remark 2.4.6. We can easily generalize the above framework to $O(\sqrt{n})$-separable graphs for which the separator can be computed in $s(n)$ time, since the only place we need such computation is to initialize or re-compute the data structure $\mathcal{D}(G, \delta)$ (after every $\Theta(\sqrt{n})$ operations). This implies that the update time will become $\tilde{O}((s(n) + n/\delta^2)/\sqrt{n})$ and the query time remains the same as before.

2.4.2 Extension to Dynamic All-Pairs Effective Resistance

We next explain how to use a dynamic ASC algorithm to obtain a fully-dynamic algorithm for maintaining an $(1+\varepsilon)$-approximation to all-pairs (resp., single-pair) effec-

Algorithm 2.6: EFFECTIVERESISTANCE(s, t)

1 ADDTERMINAL(s), ADDTERMINAL(t)
2 Let G' be the ASC of the root node G with respect to $\partial(G)$
3 Set $\psi \leftarrow$ ESTIMATEEFFRES(G', s, t)
4 Return ψ

tive resistance(s) in a $O(\sqrt{n})$-separable graph G and prove Theorem 2.1.1. The data-structure support the operations INSERT(u, v, r), DELETE(u, v), and EFFECTIVERESISTANCE(s, t) as defined in the beginning of this chapter.

Our dynamic effective resistance algorithm uses the above dynamic algorithm for maintaining a $(1 \pm \delta)$-ASC as a subroutine. Formally, to maintain $(1 + \varepsilon)$-approximations of effective resistances, we will invoke the dynamic ASC algorithm with parameters $\delta = \varepsilon/4$. To answer the queries of the effective resistance of any two given vertices, we use the following result due to Durfee et al. [88].

Theorem 2.4.7. *Fix $\delta \in (0, 1/2)$ and let $G = (V, E, \mathbf{w})$ be a weighted graph with two distinguished vertices $s, t \in V$. There is an algorithm ESTIMATEEFFRES(G, s, t) that computes a value ψ such that*

$$(1 - \delta)R^G_{\text{eff}}(s, t) \leq \psi \leq (1 + \delta)R^G_{\text{eff}}(s, t),$$

in time $\tilde{O}(m + n/\delta^2)$ with probability at least $1 - n^c$ for some constant $c \geq 1$.

For simplicity, we focus on the case that the separator of the separable graph can be computed in $\tilde{O}(n)$ time. The algorithm and analysis can be easily generalized to handle the case when the computation time for separator is $s(n, m)$, by the same argument as before.

We now describe the query operation. We first consider how to maintain all-pairs effective resistances. Given s and t, we start by calling ADDTERMINAL(s) and ADDTERMINAL(t) from the dynamic ASC data-structure. This ensures that both s and t are boundary nodes at the root node G (if they were not previously). Thus we obtain a $(1 \pm \delta)$-ASC, denoted as G', of the root node G with respect to $\partial(G)$ and run on G' a nearly linear time algorithm for estimating the $s - t$ effective resistance (see Theorem 2.4.7). Let ψ denote such an estimate. This procedure is summarized in Algorithm 2.6.

For the correctness, by Lemma 2.3.3, we have that G' preserves all-pairs effective resistances among vertices in $\partial(G)$ of G up to an $1/(1 \pm \delta) \approx (1 \pm 2\delta)$ factor. Since we ensured that s and t are included in $\partial(G)$, the $s - t$ effective resistance is approximated within the same factor. By Theorem 2.4.7, it follows that the estimate ψ approximates the effective resistance between s and t in G', up to a $(1 \pm \delta)$ factor. Combining the above guarantees, we get ψ gives an $(1 \pm 2\delta)(1 \pm \delta) \leq (1 \pm \varepsilon)$-approximation to $R^G_{\text{eff}}(s, t)$, by the choice of δ.

Once the query is answered, we then undo all the changes that we have performed in $\mathcal{T}(G)$ i.e., we bring the data-structure to its state before the query op-

eration. This ensures that the number of terminals at the root node G does not accumulate over a large sequence of query operations.

For the running time, first recall that each AddTerminal() operation can be implemented in $\tilde{O}(\sqrt{n}/\delta^2)$. Now, as $|V(G')| \leq O(\sqrt{n})$ and $|E(G')| \leq \tilde{O}(\sqrt{n}/\delta^2)$, by Theorem 2.4.7 it follows that estimate ψ can be computed in $\tilde{O}(\sqrt{n}/\delta^2)$ time. Combining the time bounds we get that that the worst-case time to answer an EffectiveResistance(s,t) query is $\tilde{O}(\sqrt{n}/\delta^2)$. Finally, note that in the same time bound, we can also undo all the changes we have made.

For the single-pair $s-t$ effective resistance, the two vertices s,t are fixed throughout all the operations. For each edge insertion or deletion, we first update the data structure in the same way as for the all-pairs version, and then we compute the $s-t$ effective resistance $R^G_{\text{eff}}(s,t)$ and store the answer. For the query for $R^G_{\text{eff}}(s,t)$, we simply report the stored answer. The update time is still $\tilde{O}(\sqrt{n}/\delta^2)$, while the query time is only $O(1)$.

2.5 Lower Bounds for Dynamic Effective Resistances

2.5.1 A Lower Bound for $O(\sqrt{n})$-Separable Graphs

In this section, we prove a conditional lower bound for incrementally or decrementally maintaining the $s-t$ effective resistance in $O(\sqrt{n})$-separable graphs and give the proof of Theorem 2.1.3. Our proof actually holds for any algorithm that maintains a $(1+O(\frac{1}{n^{36}}))$-approximation of $s-t$ effective resistance.

We first consider the incremental case, in which only edge insertions are allowed.

The reduction. We reduce the **uMv** problem (see Definition 2.2.4) with parameters $n_1 = n_2 := n_0$ to the $s-t$ effective resistance problem as follows. Let $\mathbf{M}$ be the $n_0 \times n_0$ Boolean matrix of the **uMv** problem. Let $n = n_0^2 + 2n_0 + 2$. Let $\kappa = 3(n-1)^6$.

Given the matrix $\mathbf{M}$, we construct a graph $G_{\mathbf{M}} = (V_{\mathbf{M}}, E)$ as follows.

- For each pair $1 \leq i,j \leq n_0$, we create two vertices a_{ij} and b_{ij}, and add an edge (a_{ij}, b_{ij}) if and only if $M_{ij} = 1$.
- For each row i, we create a vertex u_i and add edge (u_i, a_{ik}) for each $1 \leq k \leq n_0$. For each column j, we create a vertex v_j and add edge (v_j, b_{kj}) for each $1 \leq k \leq n_0$.

This finishes the definition of $G_{\mathbf{M}}$. Note that $V_{\mathbf{M}} = \{a_{ij}, b_{ij}, 1 \leq i,j \leq n_0\} \cup \{u_i, 1 \leq i \leq n_0\} \cup \{v_j, 1 \leq j \leq n_0\}$. For any vertex $x \in V_{\mathbf{M}}$, let $\deg_{G_{\mathbf{M}}}(x)$ denote the degree of x in $G_{\mathbf{M}}$.

Now we add two new vertices t and s to $G_{\mathbf{M}}$. For any $x \in \{a_{ij}, b_{ij}, 1 \leq i,j \leq n_0\}$, add an edge (s,x) with weight $\kappa - \deg_{G_{\mathbf{M}}}(x)$. Denote the resulting graph by G and note that G contains $|V_{\mathbf{M}} \cup \{s,t\}| = n_0^2 + 2n_0 + 2 = n$ vertices.

Assume that G is started in a dynamic effective resistance data structure. We also maintain a number of counters in the data structure. More specifically, we initialize a global counter $Y := 0$. For each vertex $x \in \{u_i, 1 \le i \le n_0\} \cup \{v_j, 1 \le j \le n_0\}$, we maintain a counter $c(x)$ which is initialized to be 0. We now explain how we use this data structure to determine $\mathbf{uMv}$.

- Once $\mathbf{u}$ arrives, for any i such that $\mathbf{u}_i = 1$, we insert an edge (t, u_i) with weight 1, increase Y and $c(u_i)$ by 1.
- Once $\mathbf{v}$ arrives, for any j such that $\mathbf{v}_j = 1$, we insert an edge (t, v_j) with weight 1, increase Y and $c(v_j)$ by 1.
- Insert an edge (s, t) with weight $\kappa - Y$. For each vertex $x \in \{u_i, 1 \le i \le n_0\} \cup \{v_j, 1 \le j \le n_0\}$, insert an edge (s, x) with weight $\kappa - c(x) - \deg_{G_{\mathbf{M}}}(x)$.
- We perform one effective resistance query $\textsc{EffectiveResistance}(s, t)$ to obtain the (approximate) $s - t$ effective resistance in the final graph. Let $\lambda = \textsc{EffectiveResistance}(s, t)$. If $\lambda \le \frac{1}{\kappa} + \frac{Y}{\kappa^3} + \frac{Y(n_0+1)}{\kappa^5} - \frac{1}{\kappa^6}$, then return 1; otherwise, return 0.

Analysis. Note that throughout the whole sequence of updates (which are only edge insertions) and queries, the dynamic graph G is always $O(\sqrt{n})$-separable, since the set $S := \{u_1, \cdots, u_{n_0}\} \cup \{v_1, \cdots, v_{n_0}\} \cup \{s, t\}$ is a balanced separator of size $O(\sqrt{n})$.

We have the following lemma that shows an important property of our reduction. The proof of the lemma is deferred to the end of this section.

Lemma 2.5.1. *For $\kappa = 3(n-1)^6$, assume that $\textsc{EffectiveResistance}(s, t)$ returns an $(1 + \frac{1}{\kappa^6})$-approximation of the $s - t$ effective resistance in the final graph G. Then the following holds:*

- *If $\mathbf{uMv} = 1$, then $\lambda \le \frac{1}{\kappa} + \frac{Y}{\kappa^3} + \frac{Y(n_0+1)}{\kappa^5} - \frac{1}{\kappa^6}$;*
- *If $\mathbf{uMv} = 0$, then $\lambda > \frac{1}{\kappa} + \frac{Y}{\kappa^3} + \frac{Y(n_0+1)}{\kappa^5} - \frac{1}{\kappa^6}$.*

Note that by the above lemma, the $\mathbf{uMv}$ problem can be solved according to our estimator λ. Thus, the lower bound for the incremental setting in Theorem 2.1.3 follows by Theorem 2.2.5 and by noting that the total number of updates is $O(n_0) = O(\sqrt{n})$ and the total number of queries is 1.

In the following we prove Lemma 2.5.1. The proof is based on a connection between the 5-length cycle detection problem and the effective resistance problem.

Proof of Lemma 2.5.1. Let G denote the final graph of our reduction. Let $H := G[V_{\mathbf{M}} \cup \{t\}]$ denote the subgraph induced by vertex set $V_{\mathbf{M}} \cup \{t\}$. We observe that in the graph H, there is a cycle of length 5 containing vertex t if and only if $\mathbf{uMv} = 1$.

On the other hand, we can use our estimator λ to distinguish if H contains a 5-length cycle incident to t or not. We let $\mathbf{A} \in \mathbb{R}^{(n-1)\times(n-1)}$ denote the adjacency matrix of the graph H. Note that all entries in A are either 1 or 0.

The first claim relates the 5-length cycle detection to the trace of a matrix related to $\mathbf{A}$. Recall that we let X_{uv} denote the entry of matrix X with row index corresponding to vertex u and column index corresponding to vertex v.

Claim 2.5.2. *Let $\mathbf{B} = \kappa \cdot \mathbf{I} - \mathbf{A}$. If H contains a 5-length cycle incident to t, then $(\mathbf{B}^{-1})_{tt} \leq \frac{1}{\kappa} + \frac{Y}{\kappa^3} + \frac{Y(n_0+1)}{\kappa^5} - \frac{1.1}{\kappa^6}$. If H does not contain a 5-length cycle incident to t, then $(\mathbf{B}^{-1})_{tt} \geq \frac{1}{\kappa} + \frac{Y}{\kappa^3} + \frac{Y(n_0+1)}{\kappa^5} - \frac{0.9}{\kappa^6}$.*

Proof. First we note that B is invertible, as it is strictly symmetric diagonally dominant. Furthermore, it holds that $\kappa \cdot \mathbf{B}^{-1} = (I - \frac{1}{\kappa} \cdot \mathbf{A})^{-1}$ and thus by the Neumann series expansion, we have

$$\kappa \cdot \mathbf{B}^{-1} = (I - \frac{1}{\kappa} \cdot \mathbf{A})^{-1} = \sum_{i=0}^{\infty} (-\frac{1}{\kappa})^i \cdot \mathbf{A}^i.$$

This further implies that

$$\begin{aligned}(\kappa \cdot \mathbf{B}^{-1})_{tt} &= \mathbf{1}_t^\top (\sum_{i=0}^{\infty} (-\frac{1}{\kappa})^i \cdot \mathbf{A}^i)\mathbf{1}_t = \sum_{i=0}^{\infty} (-\frac{1}{\kappa})^i \cdot \mathbf{1}_t^\top (\mathbf{A}^i)\mathbf{1}_t \\ &= \sum_{i=0}^{\infty} (-\frac{1}{\kappa})^i \cdot (\mathbf{A}^i)_{tt}.\end{aligned} \tag{2.3}$$

Now observe that since $\kappa = 3(n-1)^6$, the first six terms of the above power series dominate. More precisely, note that $(\mathbf{A}^i)_{tt}$ is the number of i-length paths from t to t, which is at most $(n-1)^i$. Thus

$$\sum_{i=6}^{\infty} |(-\frac{1}{\kappa})^i \cdot (\mathbf{A}^i)_{tt}| \leq \sum_{i=6}^{\infty} \frac{1}{\kappa^i} (\mathbf{A}^i)_{tt} \leq \sum_{i=6}^{\infty} \frac{1}{\kappa^i} (n-1)^i \leq \frac{0.9}{\kappa^5}.$$

Now observe that $(\mathbf{A}^0)_{tt} = \mathbf{I}_{tt} = 1$; that $\mathbf{A}_{tt} = 0$ since H is a simple graph; that $(\mathbf{A}^2)_{tt} = \deg_H(t) = Y$, where the last equation follows from the definition of Y; that $(\mathbf{A}^3)_{tt} = 0$ since there is no triangle containing t; and that $(\mathbf{A}^4)_{tt} = \sum_{w:(w,t)\in E} \sum_{x:(x,w)\in E} 1 = \sum_{w:(w,t)\in E} \deg_{G_\mathbf{M}}(w) = \det_H(t) \cdot (n_0+1) = Y(n_0+1)$. Therefore,

- If H contains a 5-length cycle incident to t, then $(\mathbf{A}^5)_{tt} \geq 2$, and thus

$$\begin{aligned}(\kappa \cdot \mathbf{B}^{-1})_{tt} &\leq 1 + \frac{Y}{\kappa^2} + \frac{Y(n_0+1)}{\kappa^4} - \frac{2}{\kappa^5} + \frac{0.9}{\kappa^5} \\ &= 1 + \frac{Y}{\kappa^2} + \frac{Y(n_0+1)}{\kappa^4} - \frac{1.1}{\kappa^5}\end{aligned}$$

- If H has no 5-length cycle incident to t, then $(\mathbf{A}^5)_{tt} = 0$, and thus

$$(\kappa \cdot \mathbf{B}^{-1})_{tt} \geq 1 + \frac{Y}{\kappa^2} + \frac{Y(n_0+1)}{\kappa^4} - \frac{0.9}{\kappa^5}$$

This completes the proof of the claim. □

The following claim relates $s-t$ effective resistance to $\mathbf{B}^{-1}$. The proof almost follows from Lemma 23 in [198], while we include a proof here for the sake of completeness.

Claim 2.5.3. *Let $\Lambda = \mathcal{E}_G(s,t)$ and $\mathbf{B} = \kappa \cdot \mathbf{I} - \mathbf{A}$. Then it holds that $\Lambda = (\mathbf{B}^{-1})_{tt}$.*

Proof. Let $\mathbf{L}$ denote the Laplacian matrix of G and let $\mathbf{v} \in \mathbb{R}^{V_{\mathbf{M}} \cup \{t\}}$ denote the vector with entries corresponding to weights between s and u for each $u \in V_{\mathbf{M}} \cup \{t\}$, i.e., $\mathbf{v}_u = \kappa - \deg_H(u)$.

Now the key observation is that

$$\mathbf{L} = \begin{pmatrix} \mathbf{B} & -\mathbf{v} \\ -\mathbf{v}^\top & \deg_G(s) \end{pmatrix}$$

For any $\mathbf{x} \in \mathbb{R}^{V_{\mathbf{M}} \cup \{t\} \cup \{s\}}$, let $\widehat{\mathbf{x}} \in \mathbb{R}^{V_{\mathbf{M}} \cup \{t\}}$ be the vector containing the first entries corresponding to vertices in $V_{\mathbf{M}} \cup \{t\}$ of $\mathbf{x}$. Let $\mathbf{y}$ be the solution of the Laplacian system $\mathbf{L}\mathbf{y} = \mathbf{1}_s - \mathbf{1}_t$. Thus, $\mathbf{y} = \mathbf{L}^\dagger(\mathbf{1}_s - \mathbf{1}_t)$. It also holds that

$$\mathbf{B} \cdot \widehat{\mathbf{y}} - \mathbf{v} \cdot y_s = -\widehat{\mathbf{1}}_t$$

In addition, we know that $\mathbf{L}\mathbf{1} = \mathbf{0}$, and thus $\mathbf{B} \cdot \widehat{\mathbf{1}} = \mathbf{v}$. This further implies that, $\widehat{\mathbf{y}} = \mathbf{B}^{-1} \cdot \mathbf{v} \cdot y_s - \mathbf{B}^{-1}\widehat{\mathbf{1}}_t = y_s \cdot \widehat{\mathbf{1}} - \mathbf{B}^{-1}\widehat{\mathbf{1}}_t$. Thus,

$$\begin{aligned}(\mathbf{1}_s - \mathbf{1}_t)^\top \mathbf{L}^\dagger (\mathbf{1}_s - \mathbf{1}_t) = (\mathbf{1}_s - \mathbf{1}_t)^\top \mathbf{y} &= y_s - \widehat{\mathbf{1}}_t^\top \cdot \widehat{\mathbf{y}} \\ &= y_s - \widehat{\mathbf{1}}_t^\top \cdot (y_s \cdot \widehat{\mathbf{1}} - \mathbf{B}^{-1}\widehat{\mathbf{1}}_t) = \widehat{\mathbf{1}}_t^\top \mathbf{B}^{-1}\widehat{\mathbf{1}}_t\end{aligned}$$

Therefore,

$$\Lambda = \mathcal{E}_G(s,t) = (\mathbf{1}_s - \mathbf{1}_t)^\top \mathbf{L}^\dagger (\mathbf{1}_s - \mathbf{1}_t) = \widehat{\mathbf{1}}_t^\top \mathbf{B}^{-1}\widehat{\mathbf{1}}_t = (\mathbf{B}^{-1})_{tt}$$

□

Finally, by the above two claims, if $\mathbf{u}\mathbf{M}\mathbf{v} = 1$, then H contains a 5-length cycle incident to t, and thus $\Lambda = (\mathbf{B}^{-1})_{tt} \leq \frac{1}{\kappa} + \frac{Y}{\kappa^3} + \frac{Y(n_0+1)}{\kappa^5} - \frac{1.1}{\kappa^6}$; if $\mathbf{u}\mathbf{M}\mathbf{v} = 0$, then H does not contain any 5-length cycle incident to t, and thus $\Lambda = (\mathbf{B}^{-1})_{tt} \geq \frac{1}{\kappa} + \frac{Y}{\kappa^3} + \frac{Y(n_0+1)}{\kappa^5} - \frac{0.9}{\kappa^6}$. The statement of the lemma then follows by the fact that λ is a $(1 + \frac{1}{\kappa^6})$-approximation of Λ, and that $\frac{1}{\kappa^6}(\frac{1}{\kappa} + \frac{Y}{\kappa^3} + \frac{Y(n_0+1)}{\kappa^5} - \frac{0.9}{\kappa^6}) < \frac{0.1}{\kappa^6}$. □

For the lower bound for the decremental setting, we start with a graph where t is initially connected to s with weight $\kappa - 2n_0$ and to all vertices $x \in \{u_i, 1 \leq i \leq n_0\} \cup \{v_j, 1 \leq j \leq n_0\}$ with weights $\kappa - 1 - \deg_{G_{\mathbf{M}}}(x)$. When the vectors $\mathbf{u}, \mathbf{v}$ arrive, we need to increase the weights of some edges (s,x) and (s,t) depending if the corresponding entry of $\mathbf{u}, \mathbf{v}$ is 1 or 0, so that every vertex in G has the same weighted degree κ. We omit further details here.

2.5.2 A Lower Bound for General Graphs

In this section, we prove Theorem 2.1.4, which gives a lower bound for incremental and decremental $s-t$ effective resistance problem in general graphs.

Proof of Theorem 2.1.4. We only consider here the incremental setting, where only edge insertions are allowed. For the decremental setting, the correctness follows from a similar construction and similar arguments for decremental lower bound in the proof of Theorem 2.1.3.

We reduce the $\mathbf{uMv}$ problem with parameters $n_1 = n_2 := n_0$ to the $s-t$ effective resistance problem as follows. Let $\mathbf{M}$ be the $n_0 \times n_0$ Boolean matrix of the $\mathbf{uMv}$ problem. Let $n = 2n_0 + 2$ and let $\kappa = 3(n-1)^5$.

We first create a bipartite graph $G_{\mathbf{M}} = ((R, C), E)$ where $R = (r_1, \cdots, r_{n_0})$ and $C = (c_1, \cdots, c_{n_0})$ corresponding to the rows and columns of $\mathbf{M}$, respectively. We add an edge (r_i, c_j) in E iff $\mathbf{M}_{ij} = 1$. This finishes the definition of $G_{\mathbf{M}}$. For each vertex $x \in R \cup C$, let $\deg_{G_{\mathbf{M}}}(x)$ denote the degree of vertex x in $G_{\mathbf{M}}$.

Now we add tow new vertices s, t to $G_{\mathbf{M}}$. Denote the resulting graph by G and note that G contains $|R \cup C \cup \{s, t\}| = 2n_0 + 2$ vertices.

Assume that G is started in a dynamic effective resistance data structure. We also initialize a global counter Y to be 0 and for each vertex $x \in R \cup C$, we initialize a counter $c(x)$ to be 0. We now explain how we use this data structure to determine $\mathbf{uMv}$.

- Once $\mathbf{u}$ arrives, for any i such that $\mathbf{u}_i = 1$, we insert an edge (t, r_i) with weight 1, and increase Y and $c(r_i)$ by 1.
- Once $\mathbf{v}$ arrives, for any j such that $\mathbf{v}_j = 1$, we insert an edge (t, c_j) with weight 1, and increase Y and $c(c_j)$ by 1.
- Insert an edge (s, t) with weight $\kappa - Y$. For each $x \in V_{\mathbf{M}}$, insert an edge (s, x) with weight $\kappa - c(x) - \deg_{G_{\mathbf{M}}}(x)$.
- We perform one effective resistance query EffectiveResistance(s, t) to obtain the (approximate) $s-t$ effective resistance in the final graph. Let $\lambda =$ EffectiveResistance(s, t). If $\lambda \leq \frac{1}{\kappa} + \frac{Y}{\kappa^3} - \frac{1}{\kappa^4}$, then return 1; otherwise, return 0.

We have the following lemma similar to Lemma 2.5.1.

Lemma 2.5.4. *For $\kappa = 3(n-1)^5$, assume that EffectiveResistance(s, t) returns a $(1 + \frac{1}{\kappa^4})$-approximation of the $s-t$ effective resistance in the final graph G. Then the following holds:*

- *If $\mathbf{uMv} = 1$, then $\lambda \leq \frac{1}{\kappa} + \frac{Y}{\kappa^3} - \frac{1}{\kappa^4}$;*
- *If $\mathbf{uMv} = 0$, then $\lambda > \frac{1}{\kappa} + \frac{Y}{\kappa^3} - \frac{1}{\kappa^4}$.*

Given the above Lemma, we can then solve the $\mathbf{uMv}$ problem according to the value of our estimator λ. Thus, the statement of the theorem follows by noting that the total number of updates is $O(n_0) = O(n)$ and the total number of queries is 1, and by Theorem 2.2.5. Now we give a sketch of the proof of the above lemma.

Proof Sketch of Lemma 2.5.4. The proof is almost the same as the proof of Lemma 2.5.1. Here we point out the main difference. Let G denote the final graph of our reduction. Let $H := G[R \cup C \cup \{t\}]$ denote the subgraph induced by vertex set $R \cup C \cup \{t\}$. We observe that in the graph H, there is a triangle incident to vertex t iff $\mathbf{uMv} = 1$. Now we use our estimator λ to distinguish if H contains a triangle incident to t or not.

We let $\mathbf{A} \in \mathbb{R}^{(n-1)\times(n-1)}$ denote the adjacency matrix of the graph H. Note that all entries in A are either 1 or 0. Let $\mathbf{B} = \kappa \cdot \mathbf{I} - \mathbf{A}$. Again, by the Neumann series expansion of $\mathbf{B}^{-1}$, we could derive the same expression of $(\kappa \cdot \mathbf{B}^{-1})_{tt}$ as Equation 2.3, that is

$$(\kappa \cdot \mathbf{B}^{-1})_{tt} = \sum_{i=0}^{\infty} (-\frac{1}{\kappa})^i \cdot (\mathbf{A}^i)_{tt}.$$

Now observe that since $\kappa = 3(n-1)^5$, the first four terms of the above power series dominate. More precisely, by the fact that $(\mathbf{A}^i)_{tt} \leq (n-1)^i$ for any $i \geq 4$, we have that

$$\sum_{i=4}^{\infty} |(-\frac{1}{\kappa})^i \cdot (\mathbf{A}^i)_{tt}| \leq \sum_{i=4}^{\infty} \frac{1}{\kappa^i} (\mathbf{A}^i)_{tt} \leq \sum_{i=4}^{\infty} \frac{1}{\kappa^i} (n-1)^i \leq \frac{0.9}{\kappa^3}.$$

Furthermore, it holds that $(\mathbf{A}^0)_{tt} = \mathbf{I}_{tt} = 1$; that $\mathbf{A}_{tt} = 0$ since H is a simple graph; and that $(\mathbf{A}^2)_{tt} = \deg_H(t) = Y$, where the last equation follows from the definition of Y. Therefore,

- If H contains a triangle incident to t, then $(\mathbf{A}^3)_{tt} \geq 2$, and thus

$$(\kappa \cdot \mathbf{B}^{-1})_{tt} \leq 1 + \frac{Y}{\kappa^2} - \frac{2}{\kappa^3} + \frac{0.9}{\kappa^3} = 1 + \frac{Y}{\kappa^2} - \frac{1.1}{\kappa^3}$$

- If H has no triangle incident to t, then $(\mathbf{A}^3)_{tt} = 0$, and thus

$$(\kappa \cdot \mathbf{B}^{-1})_{tt} \geq 1 + \frac{Y}{\kappa^2} - \frac{0.9}{\kappa^3}$$

That is, if H contains a triangle incident to t, then $(\mathbf{B}^{-1})_{tt} \leq \frac{1}{\kappa} + \frac{Y}{\kappa^3} - \frac{1.1}{\kappa^4}$. If H does not contain a triangle incident to t, then $(\mathbf{B}^{-1})_{tt} \geq \frac{1}{\kappa} + \frac{Y}{\kappa^3} - \frac{0.9}{\kappa^4}$.

Now let $\Lambda = \mathcal{E}_G(s,t)$. Then by the same argument for proving Claim 2.5.3, we have that $\Lambda = (\mathbf{B}^{-1})_{tt}$.

Finally, by the above two claims, if $\mathbf{uMv} = 1$, then H contains a triangle incident to t, and thus $\Lambda = (\mathbf{B}^{-1})_{tt} \leq \frac{1}{\kappa} + \frac{Y}{\kappa^3} - \frac{1.1}{\kappa^4}$; if $\mathbf{uMv} = 0$, then H does not

contain any triangle incident to t, and thus $\Lambda = (\mathbf{B}^{-1})_{tt} \geq \frac{1}{\kappa} + \frac{Y}{\kappa^3} - \frac{0.9}{\kappa^4}$. The statement of the lemma then follows by the fact that λ is a $(1 + \frac{1}{\kappa^4})$-approximation of Λ and that $\frac{1}{\kappa^4}(\frac{1}{\kappa} + \frac{Y}{\kappa^3} - \frac{0.9}{\kappa^4}) \leq \frac{0.1}{\kappa^4}$. □

□

2.6 Conclusion

In this chapter, we studied the problem of dynamically maintaining All-Pairs Effective Resistances in graphs that admit small separators, e.g., planar graphs. We show a fully-dynamic algorithm that reports a $(1 + \epsilon)$ approximation to any effective resistance query on a graph undergoing edge insertions and deletions with $\tilde{O}(\sqrt{n}\epsilon^{-2})$ update and query time. We also prove two conditional lower bounds, one applying to graphs with small separators and the other to general graphs, which show the hardness of the problem in the exact setting and justify our upper bounds that only support approximate queries.

Our work leaves several interesting open problems for future work. For example, it is interesting to improve upon the update query and time of our dynamic All-Pair Effective Resistances problem in planar graphs while keeping the same approximation guarantee. While we do believe that poly-logarithmic running times should be achievable for this problem, this may require developing some new ideas and techniques that go beyond the standard $\sqrt{n}$ barrier, which also appears for the dynamic planar APSP problem [9]. Another interesting direction is to extend our lower-bound for separable graphs to the more restricted family of planar graphs. Finally, the most important problem is whether there is a non-trivial fully-dynamic algorithm for maintaining All-Pairs Effective Resistances in general graphs. In Chapter 3 we make substantial progress on this question and present the first algorithm that achieves sub-linear update and query time.

CHAPTER 3

Fully Dynamic Spectral Vertex Sparsifiers and Applications

We study *dynamic* algorithms for maintaining spectral vertex sparsifiers of graphs with respect to a set of terminals K of our choice. Such objects preserve pairwise resistances, solutions to systems of linear equations, and energy of electrical flows between the terminals in K. We give a data structure that supports insertions and deletions of edges, and terminal additions, all in sublinear time. We then show the applicability of our result to the following problems.

(1) A data structure for dynamically maintaining the solutions to Laplacian systems $\mathbf{L}\mathbf{x} = \mathbf{b}$, where $\mathbf{L}$ is the graph Laplacian matrix and $\mathbf{b}$ is a demand vector. For a bounded degree, unweighted graph, we support modifications to both $\mathbf{L}$ and $\mathbf{b}$ while providing access to ϵ-approximations to the energy of routing an electrical flow with demand $\mathbf{b}$, as well as query access to entries of a vector $\tilde{\mathbf{x}}$ such that $\left\|\tilde{\mathbf{x}} - \mathbf{L}^{\dagger}\mathbf{b}\right\|_{\mathbf{L}} \leq \epsilon \left\|\mathbf{L}^{\dagger}\mathbf{b}\right\|_{\mathbf{L}}$ in $\tilde{O}(n^{11/12}\epsilon^{-5})$ expected amortized update and query time.

(2) A data structure for maintaining fully dynamic All-Pairs Effective Resistance. For an intermixed sequence of edge insertions, deletions, and resistance queries, our data structures returns $(1 \pm \epsilon)$-approximation to all the resistance queries against an oblivious adversary with high probability. Its expected amortized update and query times are $\tilde{O}(\min(m^{3/4}, n^{5/6}\epsilon^{-2})\epsilon^{-4})$ on an unweighted graph, and $\tilde{O}(n^{5/6}\epsilon^{-6})$ on weighted graphs.

The key ingredients in these results are (1) the intepretation of Schur complement as a sum of random walks, and (2) a suitable choice of terminals based on the behavior of these random walks to make sure that the majority of walks are local, even when the graph itself is highly connected and (3) maintenance of these local walks and numerical solutions using data structures.

These results together represent the first data structures for maintain key primitives from the Laplacian paradigm for graph algorithms in sublinear time without

assumptions on the underlying graph topologies. The importance of routines such as effective resistance, electrical flows, and Laplacian solvers in the static setting make us optimistic that some of our components can provide new building blocks for dynamic graph algorithms.

3.1 Introduction

Problems arising from analyzing and understanding graph structures have motivated the development of many powerful tools for storing and compressing graphs and networks. One such tool that has received a considerable amount of attention over the past two decades is graph sparsification [34, 35]. Roughly speaking, a graph sparsifier is a "compressed" version of a large input graph that preserves important properties like distance information [208], cut value [35] or graph spectrum [238]. Graph Sparsifiers fall into two main categories: *edge sparsifiers*, which are graphs that reduce the number of edges, and *vertex sparsifiers*, which are graphs that reduce the number of vertices. Both categories have many applications in approximation algorithms [100, 214], machine learning [186, 255], and most recently efficient graph algorithms [158, 188, 231, 237]. While edge sparsifiers have played an instrumental role in obtaining nearly linear time algorithms [34], their practical applicability is somewhat limited due to the fact most of the large networks are already sparse. On the other hand, vertex sparsifiers address the "real" compression of large networks by reducing the number of vertices.

While vertex sparisifers in general are significantly more difficult to generate [60, 191, 197], a notable exception is vertex sparsifiers for quadratic minimization problems, otherwise known as Schur complements. Concretely, given an undirected, weighted graph G, a subset of terminal vertices K and its corresponding Laplacian matrix, a graph H with $V(H) = K$ is a vertex resistance sparsifier of G with respect to K if the Laplacian matrix of H is obtained by the Schur complement of the Laplacian of G with respect to K. Schur complement is a central concept in physics and linear algebra with a wide range of applications including multi-grid solvers, Markov chains and finite-element analysis [85], and have also recently found extensive applications in graph algorithms [88, 89, 174, 176, 227, 228].

Most of the massive graphs in the real world, such as social networks, the web graph, are subject to frequent changes over time. This dynamic behavior of graph has been studied for several important graph problems, where the basic idea is to maintain problem solutions as graphs undergo edge insertions and deletions in time faster than recomputing the solution from scratch. Dynamic graph algorithms have also been formulated for many problems that involve edge sparsifiers [117, 138, 147, 248], as well important variants of edge sparsifiers themselves, including minimum spanning trees [139, 202, 203, 260], spanners [33], spectral sparsifiers [12], and low-stretch spanning trees [105]. However, despite the increasing importance of high quality vertex sparsifiers in graph algorithms, to the best of our knowledge very little is known about their maintenance in the dynamic setting.

In this chapter we give the first non-trival *dynamic* algorithms for maintaining Schur complements of general graphs with respect to a set of terminal of our choice. Our data-structure maintains at any point of time a $(1 \pm \epsilon)$ approximation to the Schur complement while supporting insertions and deletions of edges, and arbitrary vertex additions to the terminal set. To the best of our knowledge, prior dynamic Schur complement algorithms were only known for minor-free graphs [113, 115].

Lemma 3.1.1. *Given an error threshold $\epsilon > 0$, an unweighted undirected multi-graph $G = (V, E)$ with n vertices, m edges, a subset of terminal vertices K' and a parameter $\beta \in (0, 1)$ such that $|K'| = O(\beta m)$, there is a data-structure DynamicSC(G, K', β) for maintaining a graph $\tilde{H}$ with $\mathbf{L}_{\tilde{H}} \approx_\epsilon \mathbf{SC}(G, K)$ for some K with $K' \subseteq K$, $|K| = O(\beta m)$, while supporting $O(\beta m)$ operations in the following running times:*

- *Initialize(G, K', β): Initialize the data-structure, in $\tilde{O}(m\beta^{-2}\epsilon^{-4})$ expected amortized time.*
- *Insert(u, v): Insert the edge (u, v) to G in $\tilde{O}(1)$ amortized time.*
- *Delete(u, v): Delete the existing edge (u, v) from G in $\tilde{O}(1)$ amortized time.*
- *AddTerminal(u): Add u to K' in $\tilde{O}(1)$ amortized time.*

Our algorithm extends to weighted graphs, albeit with slightly larger running time guarantees. Concretely we give an algorithm that maintains an approximate Schur Complement with $\tilde{O}(m\beta^{-4}\epsilon^{-4})$ expected amortized time for initializing the data-structure, and $O(1)$ amortized time for the remaining operations. We discuss such extensions in Section 3.4.3.

The key algorithmic components behind the result in unweighted graphs are (1) the interpretation of Schur complement as a sum of random walks and (2) randomly picking a terminal vertex subset onto which the vertex resistance sparsifiers is constructed. Specifically, in a novel way we combine random walk based methods for generating resistance vertex sparsifiers [89] with results in combinatorics that bound the speed at which such walks spread among vertices [29]. Our result in the weighted case essentially follows the same idea except that the speed at which random walks visit different vertices in weighted networks could be very slow. To control this, we instead exploit an event driven simulation of random walks that interacts well with other parts of our data structure and leads to comparable running time guarantees.

We show the applicability of our dynamic Schur complement to two cornerstone problems in graph Laplacian literature, namely *dynamic* Laplacian solver [237] and *dynamic* All-Pair Effective Resistances [235].

Solving linear systems lies at the heart of many problems arising in scientific computing, numerical linear algebra, optimization and computer science. An important subclass of linear systems are Laplacian systems, which arise in many natural contexts, including computation of voltages and currents in electrical network. Solving Laplacian system has received increasing attention over the past years after the breakthrough work of Spielman and Teng [237] who gave the first near-linear

time algorithm. Motivated by fast Laplacian solvers in different model of computations [25, 212], we initiate the study of algorithms for dynamically solving Laplacian systems. Concretely, given a graph Laplacian $\mathbf{L} \in \mathbb{R}^{n \times n}$ and a vector $\mathbf{b} \in \mathbb{R}^n$ in the range of $\mathbf{L}$, the goal is to maintain an $\mathbf{x}$ such that $\mathbf{L}\mathbf{x} = \mathbf{b}$, while off-diagonals of $\mathbf{L}$ and the entries of $\mathbf{b}$ change over time. To allow for sub-linear query times, here we focus on querying one (or a few) coordinates of $\mathbf{x}$. Formally, given any index $u \in \{1, \ldots, n\}$, the goal is to output $\tilde{\mathbf{x}}_u$ for some approximation $\tilde{\mathbf{x}}$ of $\mathbf{L}^\dagger \mathbf{b}$. Our contribution is the first sub-linear dynamic Laplacian solver in bounded degree graphs.

Theorem 3.1.2. *For any given error threshold $m^{-1} < \epsilon < 1$, there is a data-structure for maintaining an unweighted, undirected bounded degree $G = (V, E)$ with n vertices, m edges and a vector $\mathbf{b} \in \mathbb{R}^n$ that supports the following operations in $\tilde{O}(n^{11/12}\epsilon^{-5})$ expected amortized time:*

- *INSERT(u, v): Insert the edge (u, v) with resistance 1 in G.*
- *DELETE(u, v): Delete the edge (u, v) from G.*
- *CHANGE$(u, \mathbf{b}'_u, v, \mathbf{b}'_v)$: Change $\mathbf{b}_u$ to $\mathbf{b}'_u$ and $\mathbf{b}_v$ to $\mathbf{b}'_v$ while keeping $\mathbf{b}$ in the range of $\mathbf{L}$.*
- *SOLVE(u): Return $\tilde{\mathbf{x}}_u$ with $\tilde{\mathbf{x}}$ such that $\left\|\tilde{\mathbf{x}} - \mathbf{L}^\dagger \mathbf{b}\right\|_{\mathbf{L}} \leq \epsilon \left\|\mathbf{L}^\dagger \mathbf{b}\right\|_{\mathbf{L}}$.*

Note that the $\tilde{\mathbf{x}}$ in the theorem above is not guaranteed to be inside the range of $\mathbf{L}$ and it only preserves the differences between vertices in the same connected component.

We observe that conditioning on the vector $\mathbf{b}$ having small support, i.e., a small number of non-zero elements, leads to a dynamic solver by just including the corresponding vertices into the Schur complement, and maintaining a dynamic Schur complement onto these vertices augmented with some carefully chosen additional terminals. Upon receipt of a query index, we add the corresponding vertex to the current Schur complement and simply solve a linear system there. However, note that the demand vector may have a large number of non-zero entries, thus preventing us from obtaining a sub-linear time algorithm with this approach. We alleviate this by projecting this vector onto the set of current terminals and showing that such projection can be maintained dynamically while introducing controllable error in the approximation guarantee.

Another application of our technique is dynamic maintainance of effective resistance, a well studied quantity that has direct applications in random walks, spanning trees [190] and graph sparsification [235]. We maintain (approximate) All-Pair Effective Resistances of a graph G among any pair of query vertices while supporting an intermixed sequence of edge insertions and deletions in G. Our study is also motivated in part by the wide usage of *commute distances*, a random walk-based similarity measure that has been successfully employed in important practical applications such as link predictions [183]. Since commute distance is a scaled version

of effective resistance, our dynamic algorithm readily extends to this graph measure while achieving the same approximation and running time guarantees.

Theorem 3.1.3. *For any given error threshold $\epsilon > 0$, there is a data-structure for maintaining an unweighted, undirected multi-graph $G = (V, E)$ with up to m edges that supports the following operations in $\tilde{O}(m^{3/4}\epsilon^{-4})$ expected amortized time:*

- *Insert(u, v): Insert the edge (u, v) with resistance 1 in G.*
- *Delete(u, v): Delete the edge (u, v) from G.*
- *EffectiveResistance(s, t): Return a $(1 \pm \epsilon)$-approximation to the effective resistance between s and K in the current graph G.*

Our algorithm can also handle weighted graphs, albeit with a bound of $\tilde{O}(m^{5/6}\epsilon^{-4})$ on the expected amortized update and query time. By running this algorithm on the output of a dynamic spectral sparsifier [12], we obtain a bound of $\tilde{O}(n^{5/6}\epsilon^{-6})$ per operation, which is truly sub-linear irrespective of graph density.

We are optimistic that our algorithmic ideas could be useful for dynamically maintaining a wider range of graph properties. Both the results that we give dynamic algorithms for, vertex sparsifiers and Schur complements, have wide ranges of applications in static settings, with the latter being at the core of the 'Laplacian paradigm' of graph algorithms [234, 243]. While it's less clear that solutions across multiple Laplacian solves can be propagated to each other as the input dynamically changes, repeated sparsification on the other hand represents a routine that composes and interacts well with a much wider range of primitives. As a result, we are optimistic that it can be used as a building block in dynamic versions of many existing applications of Laplacian solvers.

3.1.1 Related Works

The recent data structures for maintaining effective resistances in planar graphs [113, 115] drew direct connections between Schur complements and data structures for maintaining them in dynamic graphs. This connection is due to the preservation of effective resistances under vertex eliminations (Fact 3.2.2). From this perspective, the Schur complement can be viewed as a vertex sparsifier for preserving resistances among a set of terminal vertices.

The power of vertex or edge graph sparsifiers, which preserve certain properties while reducing problem sizes, has long been studied in data structures [95, 96]. Ideas from these results are central to recent works on offline maintenance for 3-connectivity [211], generating random spanning trees [88], and new notions of centrality for networks [182]. Our result is the first to maintain such vertex sparsifiers, specifically Schur complements, for *general* graphs in online settings.

While the ultimate goal is to dynamically maintain (approximate) minimum cuts and maximum flows, effective resistances represent a natural 'first candidate' for this direction of work due to them having perfect vertex sparsifiers. That is, for any

subset of terminals, there is a sparse graph on them that approximately preserves the effective resistances among all pairs of terminals. This is in contrast to distances, where it's not known whether such a graph can be made sparse, or in contrast to cuts, where the existence of such a dense graph is not known (assuming that we are not content with large constant or poly-logarithmic approximations).

Dynamic Graph Algorithms. The maintenance of graph properties in dynamic algorithms has been a major area of ongoing research in data structures. The problems being maintained include $2-$ or $3-$connectivity [96, 138, 139], shortest paths [10, 42, 133, 134], global minimum cut [117, 127, 177, 248], maximum matching [48, 123, 206], and maximal matching [31, 204, 233]. Perhaps most closely related to our work are dynamic algorithms that maintain properties related to paths [96, 106, 138, 147, 202, 203, 260]. In particular, the work of Wulff-Nilsen [260] also utilizes the behavior of random walks under edge deletions to keep track of low-conductance cuts.

Dynamic algorithms for evaluating algebraic functions such as matrix determinant and matrix inverse has also been considered [223]. One application of such algorithms is that they can be used to dynamically maintain single-pair effective resistance. Specifically, using the dynamic matrix inversion algorithm, one can dynamically maintain *exact* (s, t)-effective resistance in $O(n^{1.575})$ update time and $O(n^{0.575})$ query time.

Vertex Sparsifiers. Vertex sparsifiers have been studied in more general settings for preserving cuts and flows among terminal vertices [60, 171, 197]. Efficient versions of such routines have direct applications in data structures, even when they only work in restricted settings: terminal sparsifiers on quasi-bipartite graphs [24] were core routines in the data structure for maintaining flows in bipartite undirected graphs [12].

Our data structure utilizes vertex sparsifiers, but in even more limited settings as we get to control the set of vertices to sparsify onto. Specifically, the local maintenance of this sparsifier under insertions and deletions hinges upon the choice of a random subset of terminals, while vertex sparsifiers usually need to work for any subset of terminals. Evidence from numerical algorithms [89, 174] suggest this choice can significantly simplify interactions between algorithmic components. We hope this flexibility can motivate further studies of vertex sparsifiers in more restrictive, but still algorithmically useful settings.

Organization. The chapter is organized as follows. We discuss preliminaries in Section 3.2 and give an overview of the key techniques in Section 3.3. After that we give a data-structure for dynamic Schur complement on unweighted graphs in Section 3.4, which can be applied to the dynamic All-Pairs Effective Resistance problem. In Section 3.4.3, we extend our data-structure to weighted graphs. In Section 3.5, we give a data-structure for dynamic projection of a vector onto a subset of vertices of

an unweighted bounded degree graph, which we combine with dynamic Schur complement to give a dynamic Laplacian solver. In Section 3.7, we provide details on the graph approximation guarantees and properties of projections that our random walk sampling and other routines rely on. Finally, in Section 3.6, we provide an algorithm for approximately sampling the sum of reciprocals of the edge weights of a random walk which allows us to generate long random walks without going through each step.

3.2 Preliminaries

In our dynamic setting, an undirected, weighted multi-graph undergoes both insertions and deletions of edges. We let $G = (V, E, \mathbf{w})$ always refer to the *current* version of the graph. We will use n and m to denote bounds on the number of vertices and edges at any point, respectively.

For an unweighted, undirected multi-graph G, let $\mathbf{A}_G$ denote its adjacency matrix and let $\mathbf{D}_G$ its degree diagonal matrix (counting edge multiplicities for both matrices). The graph *Laplacian* $\mathbf{L}_G$ of G is then defined as $\mathbf{L}_G = \mathbf{D}_G - \mathbf{A}_G$. Let $\mathbf{L}_G^\dagger$ denote the Moore-Penrose pseudo-inverse of $\mathbf{L}_G$. We often omit the subscript when the underlying graph is clear from the context. We also need to define the indicator vector $\mathbf{1}_u \in \mathbb{R}^V$ of a vertex u such that $\mathbf{1}_u(v) = 1$ if $v = u$, and $\mathbf{1}_u(v) = 0$ otherwise. Let $\mathbf{d}(u) = \sum_{v:(u,v)\in E} \mathbf{w}(u, v)$ be the weighted degree of a vertex u. We refer the reader to Chapter 2 for definitions concerning electrical flows.

A *walk* in G is a sequence of vertices such that consecutive vertices are connected by edges. A *random walk* in G is a walk that starts at a starting vertex w_0, and at step $i \geq 1$, the vertex w_i is chosen randomly among the neighbors of w_{i-1}. If graph G is unweighted, then each of its neighbors becomes w_i with equal probability. If G is weighted, the probability $\mathbb{P}_w [w_i = u \mid w_0, \ldots, w_{i-1}]$ is proportional to the edge weight $\mathbf{w}(w_{i-1}, u)$.

Effective Resistance. For our algorithm, it will be useful to define effective resistance using linear algebraic structures. Specifically, given any two vertices u and v in G, if $\boldsymbol{\chi}(u, v) := \mathbf{1}_u - \mathbf{1}_v$, then the *effective resistance* between u and v is given by

$$R_{\text{eff}}^G(u, v) := \boldsymbol{\chi}_{u,v}^\top \mathbf{L}_G^\dagger \boldsymbol{\chi}_{u,v}.$$

Linear systems in graph Laplacian matrices can be solved in nearly-linear time [168]. One prominent application of these solvers is the approximation of effective resistances.

Lemma 3.2.1. *Fix $\epsilon \in (0, 1)$ and let $G = (V, E)$ be any graph with two arbitrary distinguished vertices u and v. There is an algorithm that computes a value ϕ such that*

$$(1 - \epsilon)R_{\text{eff}}^G(u, v) \leq \phi \leq (1 + \epsilon)R_{\text{eff}}^G(u, v),$$

in $\tilde{O}(m + n/\epsilon^2)$ time with high probability.

Schur complement. Given a graph $G = (V, E)$, we can think of the *Schur complement* as the partially eliminated state of G. This relies on some partitioning of V into two disjoint subset of vertices K, called *terminals* and $F := V \setminus K$, called *non-terminals*, which in turn partition the Laplacian $\mathbf{L}$ into 4 blocks:

$$\mathbf{L} := \begin{bmatrix} \mathbf{L}_{[F,F]} & \mathbf{L}_{[F,K]} \\ \mathbf{L}_{[K,F]} & \mathbf{L}_{[K,K]} \end{bmatrix}. \tag{3.1}$$

The *Schur complement* onto K, denoted by $\mathbf{SC}(G, K)$ is the matrix after eliminating the variables in F. Its closed form is given by

$$\mathbf{SC}(G, K) = \mathbf{L}_{[K,K]} - \mathbf{L}_{[K,F]}\mathbf{L}_{[F,F]}^{-1}\mathbf{L}_{[F,K]}.$$

It is well known that $\mathbf{SC}(G, K)$ is a Laplacian matrix of a graph on vertices in K. To simplify our exposition, we let $\mathbf{SC}(G, K)$ denote both the Laplacian and its corresponding graph. An important property of Schur complement which we exploit in this work is to view the Schur complement as a collection of random walks. This particular feature will be discussed in more detail in Section 3.3. The key role of Schur complements in our algorithms stems from the fact that they can be viewed as vertex sparsifiers that preserve pairwise effective resistances.

Fact 3.2.2 (Vertex Resistance Sparsifier)**.** *For any graph $G = (V, E)$, any subset of vertices K, and any pair of vertices $u, v \in K$,*

$$R_{\text{eff}}^{G}(u, v) = R_{\text{eff}}^{\mathbf{SC}(G,K)}(u, v).$$

Spectral Approximation

Definition 3.2.3 (Spectral Sparsifier)**.** *Given a graph $G = (V, E, \mathbf{w})$ and $\epsilon \in (0, 1)$, we say that a graph $H = (V, E', \mathbf{w}')$ is a $(1 \pm \epsilon)$-spectral sparsifier of G (abbr. $H \approx_\epsilon G$) if $E' \subseteq E$, and for all $\mathbf{x} \in \mathbb{R}^n$*

$$(1 - \varepsilon)\mathbf{x}^\top \mathbf{L}_G \mathbf{x} \leq \mathbf{x}^\top \mathbf{L}_H \mathbf{x} \leq (1 + \varepsilon)\mathbf{x}^\top \mathbf{L}_G \mathbf{x}.$$

In the dynamic setting, Abraham et al. [12] recently showed that $(1\pm\epsilon)$-spectral sparsifiers of a dynamic graph G can be maintained efficiently. This algorithm will be invoked in several occasions throughout this chapter.

Lemma 3.2.4 ([12], Theorem 4.1)**.** *Given a graph G with polynomially bounded edge weights, with high probability, we can dynamically maintain a $(1 \pm \epsilon)$-spectral sparsifier of size $\tilde{O}(n\epsilon^{-2})$ of G in $O(\log^9 n\epsilon^{-2})$ expected amortized time per edge insertion or deletion. The running time guarantees hold against an oblivious adversary.*

The above result is useful because matrix approximations also preserve approximations of their quadratic forms. As a consequence of this fact, we get the following lemma.

Lemma 3.2.5. *If H is a $(1 \pm \epsilon)$-spectral sparsifier of G, then for any pair of vertices u and v*

$$(1 - \varepsilon)R_{\text{eff}}^{G}(u, v) \leq R_{\text{eff}}^{H}(u, v) \leq (1 + \varepsilon)R_{\text{eff}}^{G}(u, v).$$

3.2.1 Projection matrix and its properties

We next define a matrix that naturally appears when performing Gaussian elimination on the non-terminal vertices. Concretely, given a graph $G = (V, E)$ and terminals $K \subseteq V$, the *matrix-projection* of the non-terminals $F = V \setminus K$ onto K is given by

$$\mathbf{P}(K) := \begin{bmatrix} -\mathbf{L}_{[K,F]}\mathbf{L}_{[F,F]}^{-1} & \mathbf{I}_K \end{bmatrix}.$$

We next review some useful properties about the matrix projection $\mathbf{P}(K)$. Consider the laplacian system $\mathbf{L}\mathbf{x} = \mathbf{b}$, where $\mathbf{L}$ is partitioned into block-matrices as in Equation (3.1). This in turn partitions the solution vector into non-terminals and terminals, i.e., $\mathbf{x} = \begin{bmatrix}\mathbf{x}_F & \mathbf{x}_K\end{bmatrix}^\top$.

Lemma 3.2.6. *Let $\mathbf{x}_K$ be a solution vector such that $\mathbf{SC}(G, K)\mathbf{x}_K = \mathbf{P}(K)\mathbf{b}$. Then there exists an extension $\mathbf{x}$ of $\mathbf{x}_K$ such that $\mathbf{L}\mathbf{x} = \mathbf{b}$.*

Proof. We assume without loss of generality that the underlying graph G is connected. Consider the following extended linear system

$$\begin{bmatrix} \mathbf{L}_{[F,F]} & \mathbf{L}_{[F,K]} \\ \mathbf{0} & \mathbf{SC}(G, K) \end{bmatrix} \begin{bmatrix} \mathbf{x}_F \\ \mathbf{x}_K \end{bmatrix} = \begin{bmatrix} \mathbf{I}_F & \mathbf{0} \\ \mathbf{P}(K) \end{bmatrix} \begin{bmatrix} \mathbf{b}_F \\ \mathbf{b}_K \end{bmatrix}$$

Using the definitions of Schur complement and projection matrix, we can rewrite the above equation as follows:

$$\begin{bmatrix} \mathbf{L}_{[F,F]} & \mathbf{L}_{[F,K]} \\ \mathbf{0} & \mathbf{L}_{[K,K]} - \mathbf{L}_{[K,F]}\mathbf{L}_{[F,F]}^{-1}\mathbf{L}_{[F,K]} \end{bmatrix} \begin{bmatrix} \mathbf{x}_F \\ \mathbf{x}_K \end{bmatrix} = \begin{bmatrix} \mathbf{I}_F & \mathbf{0} \\ -\mathbf{L}_{[K,F]}\mathbf{L}_{[F,F]}^{-1} & I_K \end{bmatrix} \begin{bmatrix} \mathbf{b}_F \\ \mathbf{b}_K \end{bmatrix}$$

Multiplying both sides from the left with

$$\begin{bmatrix} \mathbf{I}_F & \mathbf{0} \\ \mathbf{L}_{[K,F]}\mathbf{L}_{[F,F]}^{-1} & I_K \end{bmatrix},$$

we get that

$$\begin{bmatrix} \mathbf{L}_{[F,F]} & \mathbf{L}_{[F,K]} \\ \mathbf{L}_{[K,F]} & \mathbf{L}_{[K,K]} \end{bmatrix} \begin{bmatrix} \mathbf{x}_F \\ \mathbf{x}_K \end{bmatrix} = \begin{bmatrix} \mathbf{b}_F \\ \mathbf{b}_K \end{bmatrix} \text{ or } \mathbf{L}\mathbf{x} = \mathbf{b},$$

what we wanted to show.

□

The following lemma draws a connection between the projection matrix and certain probabilities which will allow us to take a combinatorial view on several cases.

Lemma 3.2.7. *Consider a graph $G = (V, E)$. For any subset of vertices $K \subseteq V$, a vertex $v \in K$, and a vertex $u \in F = V \setminus K$, let $\mathbb{P}_u\left[t_v < t_{K\setminus v}\right]$ be the probability that the random walk that starts at u hits v before hitting any other vertex from $K \setminus v$. Then we have that*

$$[\mathbf{P}(K)\mathbf{1}_u](v) = \mathbb{P}_u\left[t_v < t_{K\setminus v}\right].$$

In fact, $\{\mathbf{P}(K)\mathbf{1}_u\}_{v\in K}$ is a probability distribution for any fixed vertex $v \in F$.

Proof. First, note that if there is no path from vertices in K to $F = V \setminus K$, then the lemma holds trivially. Thus assume K and F are connected by paths. Next, let

$$\mathbf{L}_{[F,F]} = \mathbf{D}_F - \mathbf{A}_F,$$

where $\mathbf{D}_F$ is the diagonal of $\mathbf{L}_{[F,F]}$ and $\mathbf{A}_F$ is the negation of the off-diagonal entries, and then expand $\mathbf{L}_{[F,F]}^{-1}$ using the Jacobi series:

$$\begin{aligned}\mathbf{L}_{[F,F]}^{-1} &= (\mathbf{D}_F - \mathbf{A}_F)^{-1} = \mathbf{D}^{-1/2}\left(\mathbf{I} - \mathbf{D}_F^{-1/2}\mathbf{A}_F\mathbf{D}_F^{-1/2}\right)^{-1}\mathbf{D}_F^{-1/2} \\ &= \mathbf{D}_F^{-1/2}\left(\sum_{\ell=0}^{\infty}(\mathbf{D}_F^{-1/2}\mathbf{A}_F\mathbf{D}_F^{-1/2})^{\ell}\right)\mathbf{D}_F^{-1/2} = \sum_{\ell=0}^{\infty}(\mathbf{D}_F^{-1}\mathbf{A}_F)^{\ell}\mathbf{D}_F^{-1}.\end{aligned}$$

The above series converges due to the fact that $\mathbf{L}_{[F,F]}$ is strictly diagonally dominant. Concretely, the latter implies $(\mathbf{A}_F\mathbf{D}_F^{-1})^{\ell}$ tends to zero as ℓ tends to infinity. Substituting this in the definition of $\mathbf{P}(K)$ and letting $\mathbf{1}_u = \begin{bmatrix}\mathbf{1}_u^F & \mathbf{1}_u^K\end{bmatrix}^{\top}$ we get that

$$\begin{aligned}\mathbf{P}(K)\mathbf{1}_u &= \begin{bmatrix}-\sum_{\ell=0}^{\infty}\mathbf{L}_{[K,F]}(\mathbf{D}_F^{-1}\mathbf{A}_F)^{\ell}\mathbf{D}_F^{-1} & \mathbf{I}_K\end{bmatrix}\begin{bmatrix}\mathbf{1}_u^F \\ \mathbf{1}_u^K\end{bmatrix} \\ &= \sum_{\ell=0}^{\infty} -\mathbf{L}_{[K,F]}(\mathbf{D}_F^{-1}\mathbf{A}_F)^{\ell}\mathbf{D}_F^{-1}\mathbf{1}_u^F.\end{aligned}$$

In particular, it follows that for any $v \in K$

$$\left[\sum_{\ell=0}^{\infty} -\mathbf{L}_{[K,F]}(\mathbf{D}_F^{-1}\mathbf{A}_F)^{\ell}\mathbf{D}_F^{-1}\mathbf{1}_u^F\right](v) = \sum_{\substack{u_0=u,\ldots,u_{\ell-1}\in F,\\ u_\ell=v}} \frac{\prod_{i=0}^{\ell-1}\mathbf{w}(u_i,u_{i+1})}{\prod_{i=1}^{\ell-1}\mathbf{d}(u_i)}. \quad \square$$

Given a demand vector $\mathbf{b} \in \mathbb{R}^n$, we say that $\mathbf{P}(K)\cdot\mathbf{b}$ is the projection of $\mathbf{b}$ onto K. In general, the projection of $\mathbf{b}$ is shorter than the original vector $\mathbf{b}$. However, for the sake of exposition, often we consider $\mathbf{P}(K)\cdot\mathbf{b}$ to be an n-dimensional vector by assuming that all coordinates in $F = V \setminus K$ are 0.

Lemma 3.2.8. *Consider a graph $G = (V, E)$. Let $K \subseteq V$ be a subset of vertices, and let $u \in F = V \setminus K$. Consider the demand vector $\mathbf{1}_u - \mathbf{P}(K)\mathbf{1}_u$ that requests to send one unit of flow from u to K according to the probability distribution $\{\mathbf{P}(K)\mathbf{1}_u\}_{v\in K}$. Then the minimum energy needed to route this demand is given by*

$$\|\mathbf{1}_u - \mathbf{P}(K)\mathbf{1}_u\|_{\mathbf{L}^{\dagger}}^2 = (\mathbf{1}_u - \mathbf{P}(K)\mathbf{1}_u)^{\top}\mathbf{L}^{\dagger}(\mathbf{1}_u - \mathbf{P}(K)\mathbf{1}_u).$$

Proof. Given a valid demand vector $\mathbf{b}$ with $\mathbf{b}^\top \mathbf{1} = 0$, Lemma 2.1 due to Miller and Peng [194] shows that the minimum energy for routing $\mathbf{b}$ is given by $\mathbf{b}^\top \mathbf{L}^\dagger \mathbf{b}$. Since by construction we have that $[\mathbf{1}_u - \mathbf{P}(K)\mathbf{1}_u]^\top \mathbf{1} = 0$, substituting this demand vector in place of $\mathbf{b}$ gives the lemma. □

3.3 Overview

The core building block of our algorithm is a fast routine that generates and maintains an approximate Schur complement onto a set of terminals K of our choice under insertion and deletions of edges as well as terminal additions, with all of these operations being supported in sub-linear time. One of the key ideas is to view to the Schur complement as a sum of random walks, and then observe that sampling exactly one walk per edge in the original graph already yields the desired object. Concretely, we build upon ideas introduced in sparsifying random walk polynomials [66], and Schur complements [89, 174] to show that it suffices to keep a union of these walks. The following result is implicit in these works, and we review it in Section 3.7 for the sake of completeness.

Theorem 3.3.1. *Let $G = (V, E, w)$ be an undirected, weighted multi-graph with a subset of vertices K. Furthermore, let $\epsilon \in (0, 1)$, and let ρ be some parameter related to the concentration of sampling given by*

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

Let H be an initially empty graph, and for every edge $e = (u, v)$ of repeat ρ times the following procedure:

1. *Simulate a random walk starting from u until it first hits K at vertex t_1,*
2. *Simulate a random walk starting from v until it first hits K at vertex t_2,*
3. *Combine these two walks (including e) to get a walk $u = (t_1 = u_0, \ldots, u_\ell = t_2)$, where ℓ is the length of the combined walk.*
4. *Add the edge (t_1, t_2) to H with weight*

$$1/\left(\rho \sum_{i=0}^{\ell-1} (1/\mathbf{w}(u_i, u_{i+1}))\right)$$

The resulting graph H satisfies $\mathbf{L}_H \approx_\epsilon \mathbf{SC}(G, T)$ with high probability.

The output approximate Schur complement of H onto K has up to $\tilde{O}(m\epsilon^{-2})$ edges, and thus is very dense to be leveraged as a sparsifier for our applications. Fortunately, there already exist efficient dynamic spectral sparsifiers, and we can always afford to keep a sparsifier $\tilde{H}$ of H whose size is only $\tilde{O}(|K|\epsilon^{-2})$.

The performance of our data structure depends on how fast we can generate the random walks used to create H. Note that even on the length n path with terminals

K concentrated on one end, the lengths of these walks may be as long as $\Omega(n^2)$. To overcome this we shorten the walks by augmenting K with roughly $O(\beta m)$ random vertices from a carefully chosen distribution. This random augmentation of K ensures that any vertex v in G is roughly $O(\beta^{-1})$ apart from a vertex in K, and then our problem reduces to understanding the rate at which a random walk spreads among *distinct* edges. Concretely, our goal is to efficiently generate the first k distinct edges visited by a walk in G. We distinguish the following cases.

1. For unweighted graphs, we utilize a result by Barnes and Feige [29] that shows that with high probability a walk reaches k distinct edges in about k^2 steps.
2. For weighted graphs, we employ an event driven simulation of walks. Specifically, by computing the exit probability on the current set of edges visited so far, we sample the first k new edges reached by the walk in $\mathrm{poly}(k)$ time. Then, because we know the order that each edges is first reached, the first among them that belongs to K gives the intersection of the walk with K.

Following Point (1), our dynamic Schur complement data-structure H with respect to a randomly augmented K is initialized by generating for each edge $e \in E$, ρ random walk of length roughly β^{-2}. This operation costs roughly $O(m\beta^{-2})$. We then make the observation that the ability to add terminals into K means we only need to consider insertions/deletions between vertices in K. Specifically, for each affected edge we append its endpoints to K. A further advantage of this approach is that additions to K only shorten random walks in H, and the cost of shortening or truncating these random walks in H can be charged to the cost of constructing them during the initialization. Thus, it follows that we can support terminal additions, and thus insert or delete edges in $O(1)$ amortized time. Maintaining a sparsifier $\tilde{H}$ of H introduces only polylogarithmic overheads, so this step does not affect much our running times. We next discuss the applicability of this result.

The data-structure we presented readily gives a *sub-linear* dynamic Laplacian solver for the case where $\mathbf{b}$ has small support, namely fewer than βm vertices of $\mathbf{b}$ are non-zero. This can be accomplished by simply appending the entries of $\mathbf{b}$ (more precisely, their corresponding vertices) to the Schur complement H, and solving the system on H upon receipt of an index query. The resulting solution vector can then be lifted back to the original Laplacian using Lemma 3.2.6. However, note that our data-structure can only support up to $O(|K|) = O(\beta m)$ operations if we want to keep the the size of H small. Thus, to limit the growth in $|K|$ we periodically rebuild the entire data structure (i.e., we resample the set of new terminals completely) after βm operations, which in turn gives an amortized update time of $O(m\beta^{-2}/(\beta m)) = O(\beta^{-3})$. Combining this with the bound of $O(\beta m)$ on the query time we obtain the following trade-off

$$\tilde{O}(\beta^{-3} + \beta m),$$

which is minimized when $\beta = m^{-1/4}$, thus giving an update and query time of $O(m^{3/4})$.

So it remains to address the case where $\mathbf{b}$ has a large number of non-zero entries. We overcome this difficulty by projecting this vector onto the current set of terminals K using the matrix $\mathbf{P}(K)$ and analyzing the error incurred by this projection. Our main observation is that the standard notion of error in Laplacian solvers, namely the $\mathbf{L}$-norm, corresponds to energies of electrical flows. This allows us to incur error in some of the $\mathbf{b}(u)$ values and then bound the energy of fixing them. To find such flows, we once again consider our problem from a random walk perspective, namely we view the projection of $\mathbf{b}$ onto K being equivalent to moving $\mathbf{b}$ around via random walks (Lemma 3.2.7). As such walks are short on unweighted graphs, we can relate their energies to the length of the walks times $\mathbf{b}(u)^2$ (Lemma 3.2.8).

One final obstacle is that if we move some vertex u from outside of K into K, the walks affected may be from multiple $\mathbf{b}(u)$s. To address this, we bound the 'load' of a vertex, defined as the number of walks that go through it, by the total length of the walks. The latter follows from the uniform distribution of random walks being stationary. Thus, as long as we picked K so that all the entries in $V \setminus K$ have small magnitudes, each move of some u into K incurs some small error. Bounding the accumulation of such errors, and rebuilding appropriately gives the overall dynamic solver result.

One application of the dynamic Laplacian solver is that we can maintain the energy of electrical flow for routing $\mathbf{b}$. This can also be viewed as an extension of our dynamic effective resistances data-structure, which can only maintain the energy of electrical flows for $\mathbf{b}$ with two non-zeros. Some further extensions in this direction that we believe would be useful are providing implicit access to the dual electrical flows, as well as finding the k largest entries either in the flow edges or the solution vector $\mathbf{x}$. However, such extensions will likely require a better understanding of the graph sparsifier component [12], which is treated as a black box in this work.

For dynamically maintaining effective resistance in unweighted graphs, we essentially follow the same approach as with the dynamic solver for small support demand vectors, and thus obtain a running time of $O(m^{3/4})$ on both update and query time. For weighted graphs, we employ the weighted dynamic Schur complement algorithm (following Point(2)) which gives slightly weaker guarantees, namely a bound of $\tilde{O}(m^{5/6})$ on the update and query time. Interestingly, this weighted version has another immediate advantage; by running the data-structure on the output of a dynamic spectral sparsifier (Lemma 3.2.4), we obtained a bound of $\tilde{O}(n^{5/6})$ per operation, which is truly sub-linear irrespective of graph density.

3.4 Dynamic Schur Complement

In this section we show how to dynamically maintain approximate Schur complements. We first restrict our attention to unweighted graphs (i.e., prove Lemma 3.1.1), and then show how these result extend to the weighted case. We also present two applications of our data structures, namely dynamic maintenance of effective resis-

tance on both unweighted (Theorem 3.1.3) and weighted graphs (Theorem 3.4.15).

3.4.1 Dynamic Schur Complement on Unweighted Graphs

In this section we design a data-structure for maintaining approximte Schur complements under the assumption that the dynamic graph remains unweighted throughout the updates. Specifically, we have the following lemma.

Lemma 3.4.1 (Restatement of Lemma 3.1.1)**.** *Given an error threshold $\epsilon > 0$, an unweighted undirected multi-graph $G = (V, E)$ with n vertices, m edges, a subset of terminal vertices K' and a parameter $\beta \in (0, 1)$ such that $|K'| = O(\beta m)$, there is a data-structure DynamicSC(G, K', β) for maintaining a graph $\tilde{H}$ with $\mathbf{L}_{\tilde{H}} \approx_\epsilon \mathbf{SC}(G, K)$ for some K with $K' \subseteq K$, $|K| = O(\beta m)$, while supporting $O(\beta m)$ operations in the following running times:*

- *Initialize(G, K', β): Initialize the data-structure, in $\tilde{O}(m\beta^{-2}\epsilon^{-4})$ expected amortized time.*
- *Insert(u, v): Insert the edge (u, v) to G in $\tilde{O}(1)$ amortized time.*
- *Delete(u, v): Delete the existing edge (u, v) from G in $\tilde{O}(1)$ amortized time.*
- *AddTerminal(u): Add u to K' in $\tilde{O}(1)$ amortized time.*

To prove the lemma above, we first review the interpretation of Schur Complements using random walks, and then discuss how to generate and maintain these walks under edge updates and addition of terminal vertices.

Given a graph $G = (V, E)$ and a subset of terminals K recall that $\mathbf{SC}(G, K)$ was defined using an algebraic expression that involved the Laplcian of G. However, since it is still unclear how to exploit this expression in the dynamic setting we instead take a different, more 'combinatorial', view on $\mathbf{SC}(G, K)$. Concretely, we will interpret $\mathbf{SC}(G, K)$ as a collection of random walks, each starting at an edge of G and terminating in K, as described in Theorem 3.3.1.

Let H be the output graph from the construction in Theorem 3.3.1. Recall that H is an approximate Schur Complement onto K that has up to $\rho m = \tilde{O}(m\epsilon^{-2})$ edges (that is, ρ for each edge in G, where $\rho = O(\log n\epsilon^{-2})$ is the sampling parameter). As we will next show, H does not change too much (in amortized sense) upon inserting or deleting an edge in G. We will be able to maintain H such that the distribution of H is the same as $H(G)$ of the current graph G. Therefore, we can maintain these changes using a dynamic spectral sparsifier $\tilde{H}$ of H (Lemma 3.2.4), and whenever a query comes, we answer it on $\tilde{H}$ in $\tilde{O}(|K|\epsilon^{-2}) = \tilde{O}(\beta m\epsilon^{-2})$ time.

While it is widely known how to generate random walks efficiently, we note that the length of the walks generated in Theorem 3.3.1 could be prohibitively large if K is being picked arbitrarily. To see this, recall our example where we considered a path of length n with terminals K being places in one end. The length of such random walks may be as long as $\Omega(n^2)$. To shorten these random walks, we augment K' with a random subset of vertices, which results in a larger set K. Coming

Algorithm 3.1: InitializeUnweighted(G, K', β)

Input : Unweighted graph G, set of vertices $K' \subseteq V$ such that $|K'| \leq O(m\beta)$, and $\beta \in (0, 1)$
Output : Approximate Schur Complement H and union of β-shorted walks W

Set $K \leftarrow K'$, $H \leftarrow (V, \emptyset)$ and $W \leftarrow \emptyset$
For each edge $e = (u, v)$ in G, let $K \leftarrow K \cup \{u, v\}$ with probability β
Let $\rho \leftarrow O(\log n\epsilon^{-2})$ be the sampling overhead according to Theorem 3.3.1
for *each edge* $e = (u, v) \in E$ *and each* $i = 1, \ldots, \rho$ **do**
    Generate a random walk $w_1(e, i)$ from u until $\Theta(\beta^{-1} \log n)$ different edges have been hit, it reaches K, or it has hit every edge in its component
    Generate a random walk $w_2(e, i)$ from v until $\Theta(\beta^{-1} \log n)$ different edges have been hit, it reaches K, or it has hit every edge in its component
    if *both walks reach* K *at* t_1 *and* t_2 *respectively* **then**
        Connect $w_1(e, i)$, e and $w_2(e, i)$ to form a walk $w(e, i)$ between t_1 and t_2
        Let $\ell \leftarrow \ell(w_1(e, i)) + \ell(w_2(e, i)) + 1$ be the length of $w(e, i)$
        Add an edge (t_1, t_2) with weight $1/(\rho\ell)$ to H
        Add $w(e, i)$ to W
return H and W

back to the path example, βn uniformly random vertices will be roughly β^{-1} apart, and random walks will reach one of these βn vertices in about β^{-2} steps. Because G could be a multi-graph, and we want to support queries involving any vertex, we pick K as the end points of a uniform subset of edges. A case that illustrates the necessity of this choice is a path except one edge has n parallel edges. In this case it takes $\Theta(n)$ steps in expectation for a random walk to move away from the end points of that edge. This choice of K completes the definition of our data structure, which we summarize in Algorithm 3.1, and will discuss throughout the rest of this section.

The performance of our data structures hinge upon the properties of the random walks generated. We start by formalizing such a structure involving the set of augmented terminals described above while parameterizing it with a more general probability β for including the endpoints of the edges.

Definition 3.4.2 (β-shorted walks)**.** *Let G be an weighted, undirected multi-graph and $\beta \in (0, 1)$ a parameter. A collection of β-shorted walks W on G is a set of random walks created as follows:*

1. *Choose a subset of terminal vertices K, obtained by including the endpoints of each edge independently with probability at β.*
2. *For each edge $e \in E$, generate ρ walks from its endpoints either until $\Omega(\beta^{-1} \log n)$ different edges have been hit, or they reach K, or they visited each edge that is in the same connected component as e.*

As we will shortly seee, the main property of the collection W is that its random walks are short. Moreover, we will also prove that all walks in W will reach K with high probability. These guarantees are summarized in the following theorem.

Theorem 3.4.3. *Let $G = (V, E)$ be any undirected multi-graph, and $\beta \in (0, 1)$ a parameter. Any set of β-shorted walks W, as described in Definition 3.4.2, satisfies the following:*

- *With high probability, any random walk in W starting in a connected component containing a vertex from K terminates at a vertex in K.*

Note that Theorem 3.4.3 is conditioned upon the connected component having a vertex in K: this is necessary because walks stay inside a connected component. However, this does not affect our queries: our data-structure has an operation for making any vertex u a terminal, which we call during each query to ensure both s and K are terminal vertices. Such an operation interacts well with Theorem 3.4.3 because it can only increase the probability of an edge's endpoints being chosen.

Proving the theorem requires to determine the rate at which a random walk visits at least $\beta^{-1} \log n$ edges. It turns out that a random walk of length $\tilde{O}(\beta^{-2})$ is highly likely to achieve this. For formally showing this, we need the following result by Barnes and Feige [29].

Theorem 3.4.4 ([29], Theorem 1.2)**.** *There is an absolute constant c_{BF} such that for any undirected, unweighted, multi-graph G with n vertices and m edges, any vertex u and any value $\hat{m} \leq m$, the expected time for a random walk starting from u to visit at least $\hat{m}$ distinct edges is at most $c_{BF}\hat{m}^2$.*

The above theorem can be amplified into a with high probability bound by repeating the walk $O(\log n)$ times.

Corollary 3.4.5. *In any undirected unweighted multi-graph G with m edges, for any starting vertex u, any length ℓ, and a parameter $\delta \geq 1$, a walk of length $c_{BF} \cdot \delta \cdot \ell \log n$ from u visits at least $\ell^{1/2}$ distinct edges with probability at least $1 - n^{-\delta}$.*

Proof. We can view each such walk as a concatenation of $\delta \log n$ sub-walks, each of length $c_{BF} \cdot \ell$.

We call a sub-walk *good* if the number of distinct edges that it visits is at least $\ell^{1/2}$. Applying Markov's inequality to the result of Theorem 3.4.4, a walk takes more than $O(\ell)$ steps to visit $\ell^{1/2}$ distinct edges with probability at most $1/2$.

This means that each subwalk fails to be good with probability at most $1/2$. Thus, the probability that all subwalks fail to be good is at most $2^{-\delta \log n} = n^{-\delta}$. The result then follows from an union bound over all starting vertices $u \in V$. □

We now have all the tools to prove Theorem 3.4.3.

Proof of Theorem 3.4.3. For any walk w, we define $V(w)$ (respectively, $E(w)$) to be the set of distinct vertices (respectively, edges) that a walk w visits. Consider a random walk w that starts at u of length

$$\ell = c_{BF} \cdot \delta^3 \cdot \beta^{-2} \log^3 n$$

where δ is a constant related to the success probability.

If the connected component containing the walk has fewer than

$$\delta \cdot \beta^{-1} \cdot \log n$$

edges, then Corollary 3.4.5 gives that we have covered this entire component with high probability, and the guarantee follows from the assumption that this component contains a vertex of K.

Otherwise, we will show that w reached enough edges for one of their endpoints to be picked to be picked into K with high probability. The key observation is that because w is generated independently from K, we can bound the probability of this walk not hitting K by first generating w, and then K. Specifically, for any size threshold z, we have

$$\begin{aligned} \mathbb{P}_{K,w}\left[V\left(w\right) \cap K = \emptyset\right] &= \mathbb{P}_{w,K}\left[V\left(w\right) \cap K = \emptyset\right] \\ &\leq \mathbb{P}_{w}\left[\left|E\left(w\right)\right| \leq z\right] + \mathbb{P}_{w:\left|E(w)\right| \geq z, K}\left[V\left(w\right) \cap K = \emptyset\right]. \end{aligned} \tag{3.2}$$

By Corollary 3.4.5 and the choice of ℓ, if we set

$$z = \delta \cdot \beta^{-1} \cdot \log n,$$

then the first term in Equation (3.2) is bounded by $n^{-\delta}$. For bounding the second term, we can now focus on a particular walk $\widehat{w}$ that visits at least $\delta \cdot \beta^{-1} \cdot \log n$ distinct edges, i.e.,

$$\left|E\left(\widehat{w}\right)\right| \geq \delta \cdot \beta^{-1} \log n.$$

Recall that we independently added the end points of each of these edges into K with probability β. If any of them is selected, we have a vertex that is both in $V(\widehat{w})$ and K. Thus the probability that K contains no vertices from $V(\widehat{w})$ is at most

$$(1-\beta)^{\left|E(\widehat{w})\right|} \leq (1-\beta)^{\delta \cdot \beta^{-1} \log n} \leq e^{-\delta \log n} \leq n^{-\delta},$$

which completes the proof. □

Corollary 3.4.5 together with Theorem 3.4.3 yield the following lemma.

Lemma 3.4.6. *Algorithm 3.1 runs in $\tilde{O}(m\beta^{-2}\epsilon^{-2})$ time and outputs a graph H with $\mathbf{L}_H \approx_\epsilon \mathbf{SC}(G, K)$, with high probability.*

Proof. By Corollary 3.4.5, the length of each walk generated in Algorithm 3.1 is bounded by $O(\beta^{-2}\log^3 n)$. In addition, note that each step in a random walk can be simulated in $O(1)$ time. This is due to the fact that we can sample an integer in $[0, n-1]$ by drawing $x \in [0,1]$ uniformly and taking $\lfloor xn \rfloor$. Combining these with the fact that the algorithm generates $\rho m = \tilde{O}(m\epsilon^{-2})$ walks, it follows that the running time of the algorithm is dominated by $\tilde{O}(m\beta^{-2}\epsilon^{-2})$.

Note that the collection of generated walks form the set W of β-shorted walks. By Theorem 3.4.3, with high probability, each of the walks that starts at a component containing a vertex in K hits K. Conditioning on the latter, Theorem 3.3.1 gives that with high probability, $\mathbf{L}_H \approx_\epsilon \mathbf{SC}(G, K)$. □

Handling edge updates and terminal additions. We start by observing that there is always a one-to-one correspondence between the collection of β-shorted walks W and our approximate Schur complement H. Accordingly, our primary concern will be supporting the INSERT, DELETE, and ADDTERMINAL operations in the collection W. However, as W undergoes changes, we need to efficiently update the sparsifier H. To handle these updates, we would ideally have efficient access to which walks in W are affected by the corresponding updates.

To achieve this, we index into walks that utilize a vertex or an edge, and thus set up a reverse data structure pointing from vertices and edges to the walks that contain them. The following lemma says that we can modify this representation with minimal cost.

Lemma 3.4.7. *For the collection of β-shorted walks W, let W_v and W_e be the specific walks of W that contain vertex v and edge e, respectively. We can maintain a data structure for W such that for any vertex v or edge e it reports either*

- *All walks in W_v or W_e in $O(|W_v|)$ or $O(|W_e|)$ time, respectively, or*

with an additional $O(\log n)$ overhead for any changes made to W.

Proof. For every vertex (respectively, edge), we can maintain a balanced binary search tree consisting of all the walks that use it in time proportional to the number of vertices (respectively, edges) in the walks. Supporting rank and select operations on such trees then gives the claimed bound. □

As a result, any update made to the collection of walks can be updated in the approximate Schur complement H generated from these walks in $O(\log n)$ time. We now have all the necessary ingredients to prove Lemma 3.1.1.

Proof of Lemma 3.1.1. We give a two-level data-structure for dynamically maintaining Schur complements. Specifically, we keep the terminal set K of size $\Theta(m\beta)$. This entails maintaining

1. an approximate Schur complement H of G with respect to K (Theorem 3.3.1),
2. a dynamic spectral sparsifier $\tilde{H}$ of H (Lemma 3.2.4).

Algorithm 3.2: AddTerminal(u)

Input : Vertex u such that $u \notin K$

1 Set $K \leftarrow K \cup \{u\}$

2 Shorten all random walks in W to the first location they meet u

3 Update the corresponding edges in H and $\tilde{H}$

We implement the procedure Initialize by running Algorithm 3.1, which produces a graph H and then computing a spectral sparsifier $\tilde{H}$ of H using Lemma 3.2.4. Note that by construction of our data-structure, every update in H will be handled by the black-box dynamic sparsifier $\tilde{H}$.

As we will shortly see, operations Insert and Delete will be reduced to adding terminals to the set K. Thus, the bulk of our effort is devoted to implementing the procedure AddTerminal. Let u be a non-terminal vertex that we want to append to K. We set $K \leftarrow K \cup \{u\}$, and then shorten all the walks at the first location they meet u. This shortening of walks induces in turn edge insertions and deletions to H, which are then processed by $\tilde{H}$. The pseudocode for this operation is summarized in Algorithm 3.2. To quickly locate the first appearances of u in the random walks from W, we make use of the data-structure from Lemma 3.4.7. Let us first describe the construction of such data-structure during the preprocessing phase. Let W_u be the balanced binary search tree consisting of all the walks that use the vertex u in W. Fix $w \in W_u$. For any $t \geq 0$, if w visits u after K steps, we check whether W_u contains w or not. If the latter holds, we know that u has appeared before in w and we do not need to add w to W_u. Otherwise, we add w to W_u as this is the first time the walk w visits u. After locating the first appearances of u, we cut the walks in these locations, delete the corresponding affected walks (together with their weight from H), and insert the new shorter walks to H. Note that we can simply use arrays to represent each random walk in W.

We next discuss the implementation of operations Insert and Delete. Specifically, upon insertion or deletion of an edge $e = (u, v)$ in G, we append both u and v to the terminal set K. Now, all the walks that pass through u or v in W must be shorten at the first location they meet u or v. For inserting an edge (u, v) with weight $\mathbf{w}(u, v)$ in G, we simply add ρ trivial random walks (i.e., the edge (u, v)) of weight $\frac{\mathbf{w}(u,v)}{\rho}$ to H (which sum up to the edge (u, v) itself). For deleting the edge (u, v) with weight $\mathbf{w}(u, v)$ from G, simply delete these ρ random walks between u and v in H (which exist since we guaranteed that u and v are added as terminals to H).

We next analyze the performance of our data-structure. Let us start with the pre-processing time. First, by Lemma 3.4.6 we get that the cost for constructing H on a graph with m edges is bounded by $\tilde{O}(m\beta^{-2}\epsilon^{-2})$. Next, since H has $\tilde{O}(m\epsilon^{-2})$ edges, constructing $\tilde{H}$ takes $\tilde{O}(m\epsilon^{-4})$ time. Thus, the amortized time of Initialize operation is bounded by $\tilde{O}(m\beta^{-2}\epsilon^{-4})$.

We now analyze the update operations. By the above discussion, note that it

suffices to bound the time for adding a vertex to K, which in turn (asymptotically) bounds the update time for edge insertions and deletions. The main observation we make is that adding a vertex to K only shortens the existing walks, and Lemma 3.4.7 allows us to find such walks in time proportional to the amount of edges deleted from the walk. Since the walk needed to be generated in the INITIALIZE operation, the deletion of these edges take equivalent time to generating them. Moreover, we note that (1) handling the updates in $\tilde{H}$ induced by H introduces additional $O(\text{poly}(\log n)\epsilon^{-2})$ overheads, and (2) adding or deleting ρ edges until the next rebuild costs $\tilde{O}(\beta m\epsilon^{-2})$, since we process only up to βm operations. These together imply that the amortized cost for adding a terminal can be charged against the preprocessing time, which is bounded by $\tilde{O}(m\beta^{-2}\epsilon^{-4})$, up to poly-logarithmic factors. Thus it follows that the operations ADDTERMINAL, INSERT and DELETE can be implemented in $\tilde{O}(1)$ amortized update time. □

3.4.2 Dynamic All-Pair Effective Resistance on Unweighted Graphs

In this section we present the first application of our dynamic Schur complement data structure for unweighted graphs. Concretely, we design a dynamic algorithm that supports an intermixed sequence of edge insertions, deletions and pair-wise resistance queries, and returns a $(1 \pm \epsilon)$-approximation to all the resistance queries.

We start by reviewing two natural attempts for solving this problem.

- First, since spectral sparsifiers preserve effective resistances (Lemma 3.2.5), we could dynamically maintain a spectral sparsifier (Lemma 3.2.4), and then compute the (s, t) effective resistance on this sparsifier. This leads to a data structure with $\text{poly}(\log n, \epsilon^{-1})$ update time and $\tilde{O}(n\epsilon^{-2})$ query time.
- Second, by the preservation of effective resistances under Schur complements (Fact 3.2.2), we could also utilize Schur complements to obtain a faster query time among a set of βm terminals, K, for some reduction factor $\beta \in (0, 1)$, at the expense of a slower update time. Specifically, after each edge update, we recompute an approximate Schur complement of the sparsifier onto K in time $\tilde{O}(m\epsilon^{-2})$ [88], after which each query takes $\tilde{O}(\beta m\epsilon^{-2})$ time.

The first approach obtains sublinear update time, while the second approach gives sublinear query time. Our algorithm stems from combining these two methods, with the key additional observation being that adding more vertices to K makes the Schur complement algorithm more local. Specifically, using Lemma 3.1.1 leads to a data-structure for dynamically maintaining all-pair effective resistances.

Proof of Theorem 3.1.3. Let $\mathcal{D}(\tilde{H})$ denote the data structure that maintains a dynamic (sparse) Schur complement $\tilde{H}$ of G (Lemma 3.1.1). Since $\mathcal{D}(\tilde{H})$ supports only up to βm operations, we rebuild $\mathcal{D}(\tilde{H})$ on the current graph G after such many operations. Note that the operations INSERT and DELETE on G are simply passed to $\mathcal{D}(\tilde{H})$. For processing the query operation EFFECTIVERESISTANCE(s, t), we declare

s and t terminals (using the operation AddTerminal of $\mathcal{D}(\tilde{H})$), which ensures that they are both now contained in $\tilde{H}$. Finally, we compute the (approximate) effective resistance between s and t in $\tilde{H}$ using Lemma 3.2.1.

We now analyze the performance of our data-structure. Recall that the insertion or deletion of an edge in G can be supported in $\tilde{O}(1)$ expected amortized time by $\mathcal{D}(\tilde{H})$. Since our data-structure is rebuilt every βm operations, and rebuilding $\mathcal{D}(\tilde{H})$ can be implemented in $\tilde{O}(m\beta^{-2}\epsilon^{-4})$, it follows that the amortized cost per edge insertion or deletion is

$$\frac{\tilde{O}(m\beta^{-2}\epsilon^{-4})}{\beta m} = \tilde{O}(\beta^{-3}\epsilon^{-4}).$$

The cost of any (s,t) query is dominated by (1) the cost of declaring s and t terminals and (2) the cost of computing the (s,t) effective resistance to ϵ accuracy on the graph $\tilde{H}$. Since (1) can be performed in $\tilde{O}(1)$ time, we only need to analyze (2). We do so by first giving a bound on the size of K. To this end, note that each of the m edges in the current graph adds two vertices to K with probability β independently. By a Chernoff bound, the number of random augmentations added to K is at most $2\beta m$ with high probability. In addition, since $\mathcal{D}(\tilde{H})$ is rebuilt every βm operations, the size of K never exceeds $4\beta m$ with high probability. The latter also bounds the size of $\tilde{H}$ by $\tilde{O}(\beta m\epsilon^{-2})$ and gives that the query cost is $\tilde{O}(\beta m\epsilon^{-4})$.

Combining the above bounds on the update and query time, we obtain the following trade-off

$$\tilde{O}\left((\beta m + \beta^{-3})\epsilon^{-4}\right),$$

which is minimized when $\beta = m^{-1/4}$, thus giving an expected amortized update and query time of

$$\tilde{O}(m^{3/4}\epsilon^{-4}). \qquad \square$$

3.4.3 Dynamic Schur Complement on Weighted Graphs

In this section we present an extension of Lemma 3.1.1 to weighted graphs while slightly increasing the running time guarantees. Concretely, we prove the following lemma.

Lemma 3.4.8. *Given an error threshold $\epsilon > 0$, a weighted, undirected multigraph $G = (V, E, \mathbf{w})$ with n vertices, m edges, a subset of terminal vertices K' and a parameter $\beta \in (0,1)$ such that $|K'| = O(\beta m)$, there is a data-structure WeightedDynamicSC(G, K', β) for maintaining a graph $\tilde{H}$ with $\mathbf{L}_{\tilde{H}} \approx_\epsilon \mathbf{SC}(G, K)$ for some K with $K' \subseteq K$, $|K| = O(\beta m)$, while supporting $O(\beta m)$ operations in the following running times:*

1. Initialize(G, K', β)*: Initialize the data-structure in $\tilde{O}(m\beta^{-4}\epsilon^{-4})$ expected amortized time.*
2. Insert(u, v, w)*: Insert the edge (u, v) with weight w to G in $\tilde{O}(1)$ amortized time.*

3. *DELETE(u, v): Delete the existing edge (u, v) from G in $\tilde{O}(1)$ amortized time.*
4. *ADDTERMINAL(u): Add u to K' in $\tilde{O}(1)$ amortized time.*

While the extension of our data-structure to weighted graphs builds upon the ideas we used in the unweighted case, there are a few obstacles that force us to introduce new components in our algorithm in order to make such an extension feasible. To illustrate, consider path of constant length with edge weights alternating between 1 and n^{10}. Recall that the running time our data-structure depends on the speed at which random walks visit distinct edges in a graph. Due to the structure of the edge weights, a random walk in this graph is expected to take $\Theta(n^{10})$ steps before hitting a constant number of different edges. This shows that the naive generation of random walks in weighted graphs may be computationally prohibitive for our purposes.

To rectify the above issue, we make the important observation that it is not necessary to keep information for every single step of a random walk. Instead, it would suffice if we could efficiently determine the step at which the walk meets a new vertex along with the corresponding weight associated with the walk, which defines the edge weight that is added to the sparsifier. This high-level idea allows us to generate random walks much faster, and we next make this more precise.

Following the notation we used in the unweighted case, for an arbitrary vertex $v \in V$, a set of terminals $K \subseteq V$ and a parameter $\beta \in (0, 1)$, a *β-shorted walk* with respect to v and K is a random walk that starts at a given vertex $v \in V$ and halts whenever $\Omega(\beta^{-1} \log n)$ *different* vertices have been hit, it reaches a vertex in K, or it has hit every edge in the connected component containing v. The main contribution of this section is summarized in the following lemma.

Lemma 3.4.9. *Let $G = (V, E, \mathbf{w})$ be an undirected, weighted graph with polynomially bounded weights. Let $K \subseteq V$ be a set of terminals and $v \in V$ be an arbitrary vertex. Then there is an algorithm that generates a β-shorted random walk with respect to v and K and approximates its corresponding weight up to a $(1 + \epsilon)$ multiplicative error in $\tilde{O}(\beta^{-4}\epsilon^{-2})$ time.*

We first give an intuition behind the algorithm in the above lemma and briefly describe how this algorithm interacts with other parts of our dynamic data-structure. Let $w = (w_0, \ldots, w_t)$ be a random walk that starts at an endpoint of an edge, and define

$$s(w) := \sum_{i=1}^{t} \frac{1}{\mathbf{w}(w_{i-1}, w_i)}, \tag{3.3}$$

to be its corresponding weight. Recall that before adding the walk w to H, we must scale it proportionally to $1/s(w)$ (Theorem 3.3.1). Observe that throughout our dynamic algorithm, the only modification we might do to w is to truncate it at the first location it meets a new vertex u that is being declared a terminal. Moreover, after this modification, note that the old value of $s(w)$ is no longer valid and we need to extract $s(w)$ that corresponds to the new walk. To allow efficient access to such

information, we can view the walk w as being split into sub-walk segments by the first locations w meets new vertices and store the weights of each such sub-walks. As we will next see, this bookkeeping alone allows us to proceed with the same algorithm as in the unweighted case.

We next give the three main components for implementing the algorithm stated in Lemma 3.4.9.

(A) Sample the number of steps needed for a random walk w to visit a new vertex.

(B) Sample a new *distinct* vertex that w hits, and its corresponding edge.

(C) Sample the (approximate) weight of a random walk between two given vertices.

After describing each of them, we will see that their combination naturally leads to our desired result.

Let us first discuss (A). For any $t \geq 0$, consider a t-step random walk w and let $U = \{w_0, \ldots, w_t\}$ be the set of *distinct* vertices that w has visited up to step t. Define $u := w_t \in U$ to be the *current* vertex of the walk w. Our goal is to efficiently sample the number of steps the walk w needs to visit a vertex not in U. To this end, we start by introducing some useful notation. For any $i \geq 0$, let $p^{\text{new}}(i)$ be the probability that w meets a new vertex that is not in U in $w_{t+1}, \ldots, w_{t+i}$. For $v \in U$, let $\mathbf{p}_i(v)$ be the probability that $w_{t+i} = v$, conditioned on w not having met any new vertex in $w_{t+1}, \ldots, w_{t+i-1}$. Then it can be easily verified that both $p^{\text{new}}(i)$ and $\mathbf{p}_i(v)$ are just linear combinations of $p^{\text{new}}(i)$ and $\mathbf{p}_{i-1}(v)$

$$\mathbf{p}_i(v) = \sum_{u \in U} \left(\mathbf{p}_{i-1}(u) \cdot \frac{\mathbf{w}(u,v)}{\mathbf{d}(u)} \right), \quad \forall v \in U. \tag{3.4}$$

$$p^{\text{new}}(i) = p^{\text{new}}(i-1) + \sum_{u \in V \setminus U} \sum_{v \in U} \left(\mathbf{p}_{i-1}(v) \cdot \frac{\mathbf{w}(v,u)}{\mathbf{d}(v)} \right). \tag{3.5}$$

Next, using the linearity of the recurrences in (3.5) and (3.4) we can find a matrix $\mathbf{W}$ of dimension $(k+1) \times (k+1)$, where $k = |U|$, satisfying the following equality

$$\begin{bmatrix} \mathbf{p}_i \\ p^{\text{new}}(i) \end{bmatrix} = \mathbf{W} \cdot \begin{bmatrix} \mathbf{p}_{i-1} \\ p^{\text{new}}(i-1) \end{bmatrix}, \quad \forall i \geq 1. \tag{3.6}$$

The main advantage introducing such a matrix is that it allows us to efficiently compute $p^{\text{new}}(i)$ and $\mathbf{p}_i$ using fast exponentiation via repeated squaring. Specifically, let $\mathbf{p}_0$ be a unit vector of dimension k, where for the current vertex u of the walk w we have that $\mathbf{p}_0(u) = 1$, and 0 otherwise. Let $\hat{\mathbf{p}}_0 = [\mathbf{p}_0 \quad p^{\text{new}}(0)]^\top$ be the extended $k+1$ dimension vector, where $p^{\text{new}}(0) = 0$. For any $i \geq 1$, repeatedly applying Equation (3.6) and letting $\hat{\mathbf{p}}_i := \mathbf{W}^i \hat{\mathbf{p}}_0$ yields

$$\hat{\mathbf{p}}_i(v) = \mathbf{p}_i(v), \ \forall v \in U \quad \text{and} \quad \hat{\mathbf{p}}_i(k+1) = p^{\text{new}}(i). \tag{3.7}$$

Algorithm 3.3: BINARYSEARCH($\mathbf{W}, \hat{\mathbf{p}}_0, M$)

Input : A $(k+1) \times (k+1)$ matrix $\mathbf{W}$, a $(k+1)$ dimensional vector $\hat{\mathbf{p}}_0$, and an integer M

Output : An integer

Set $\ell \leftarrow 0$, $r \leftarrow M$, $\ell p \leftarrow 0$ and $rp \leftarrow 1$

while $(\ell \neq r)$ **do**

  Set $\eta \leftarrow \lfloor (\ell + r)/2 \rfloor$

  Compute $\hat{\mathbf{p}}_\eta = \mathbf{W}^\eta \hat{\mathbf{p}}_0$ using Lemma 3.4.10

  Set $p^{\text{new}}(\eta) = \hat{\mathbf{p}}_\eta(k+1)$

  With probability $(p^{\text{new}}(\eta) - \ell p)/(rp - \ell p)$, set $r \leftarrow \eta$, and $rp = p^{\text{new}}(\eta)$, otherwise, with probability $(rp - p^{\text{new}}(\eta))/(rp - \ell p)$, set $\ell \leftarrow \eta + 1$, $\ell p = p^{\text{new}}(\eta)$

return ℓ

Using the above relation, we can use fast exponentiation via repeated squaring to compute $p^{\text{new}}(i)$ in $O(k^3 \log(i))$ time. This follows directly from the following well-known lemma, which we will exploit in a few other places throughout this work.

Lemma 3.4.10. *Let $\mathbf{B}$ be a matrix of dimension $n \times n$, and $\mathbf{B}^i$ denote the i-th power of $\mathbf{B}$, for any $i \geq 1$. Then there is an algorithm that computes $\mathbf{B}^i$ in $O(n^3 \log(i))$ time.*

We now have all the tools to describe the sampling procedure for computing the number of steps that the walk needs to visit a vertex that is distinct from the vertices in U. We accomplish this using a "binary search"-inspired subroutine, which works as follows. As an input, our algorithm is given a $(k+1) \times (k+1)$ matrix $\mathbf{W}$ (as defined in Equation (3.6)), the vector $\hat{\mathbf{p}}_0$, and an integer M, which is an upper-bound on the cover time of G. The algorithm also maintains variables $\ell, r, \ell p, rp$ with the following initialization $\ell \leftarrow 0$, $r \leftarrow M$, $\ell p \leftarrow 0$ and $rp \leftarrow 1$. As long as $(\ell \neq r)$, it defines the average $\eta = \lfloor (\ell + r)/2 \rfloor$ and then proceeds to compute $\hat{\mathbf{p}}_\eta = \mathbf{W}^\eta \hat{\mathbf{p}}_0$ using Lemma 3.4.10. Note that $p^{\text{new}}(\eta) = \hat{\mathbf{p}}_\eta(k+1)$ by Equation (3.7). Finally, the algorithm uses $p^{\text{new}}(\eta)$ to randomly decide whether w meets a new vertex in the next η steps or not. In other words, it updates the maintained variables using the rule below:

1. with probability $(p^{\text{new}}(\eta) - \ell p)/(rp - \ell p)$, set $r \leftarrow \eta$, and $rp = p^{\text{new}}(\eta)$,
2. otherwise, with probability $(rp - p^{\text{new}}(\eta))/(rp - \ell p)$, set $\ell \leftarrow \eta + 1$, $\ell p = p^{\text{new}}(\eta)$.

If $(\ell = r)$, then the algorithm returns ℓ. This procedure is summarized in Algorithm 3.3.

We next show the correctness of the above procedure. To do so, we first need the following notation. For a t-step random walk w and a current vertex $u = w_t \in U$, let $X(u, U)$ be the smallest number of steps of steps needed for w to visit a vertex

not in U, i.e., $X(u,U) = \min\{i \mid i \geq 1,\ w_{t+i} \notin U\}$. Note that $X(u,U)$ is a random variable, and $X(u,U) \leq M$ by definition of M.

Lemma 3.4.11. *Let w be a t-step random walk, U the set of distinct vertices w visited, $u = w_t \in U$ the current vertex and $k = |U|$ the number of distinct vertices w has visited so far. For $\mathbf{W}$, $\hat{\mathbf{p}}_0$, and M defined as above, BinarySearch$(\mathbf{W}, \hat{\mathbf{p}}_0, M)$ correctly samples $X(u,U)$, i.e., the number of steps w needs to visit a vertex not in U, in $O(k^3 \log^2 M)$ time.*

Proof. By Equation (3.7) and Line 4 in Algorithm 3.3, note that $p^{\text{new}}(\eta)$ is the probability that w meets a new vertex in the fist η steps. The correctness of BinarySearch can be proven using an inductive argument on the number of iterations of the while loop. Here, we just show the crucial parts for being able to apply such an argument. First, observe that right after Line 2 in the while loop, we have that

$$(rp - \ell p) = \mathbb{P}_{X(u,U)}\left[\ell \leq X(u,U) \leq r\right].$$

The latter holds because $\ell p = \mathbb{P}_{X(u,U)}\left[X(u,U) \leq \ell\right]$ and $rp = \mathbb{P}_{X(u,U)}\left[X(u,U) \leq r\right]$, which in turn can be verified for each assignment of ℓ and r. Next, we prove that conditioning on $\ell \leq X(u,U) \leq r$ right after Line 2 in the while loop, Algorithm 3.3 samples $X(u,U)$ from the correct distribution. This is true when $(\ell = r)$, since the condition of the while loop is no longer satisfied and the algorithm returns ℓ. If, however $(\ell \neq r)$, then we need to compute the following probabilities: (1) $\mathbb{P}_{X(u,U)}\left[X(u,U) \leq \eta \mid \ell \leq X(u,U) \leq r\right]$ and (2) $\mathbb{P}_{X(u,U)}\left[X(u,U) > \eta \mid \ell \leq X(u,U) \leq r\right]$. To determine (1), we get that

$$\begin{aligned}
&\mathbb{P}_{X(u,U)}\left[X(u,U) \leq \eta \mid \ell \leq X(u,U) \leq r\right] \\
&= \frac{\mathbb{P}_{X(u,U)}\left[(X(u,U) \leq \eta) \wedge (\ell \leq X(u,U) \leq r)\right]}{\mathbb{P}_{X(u,U)}\left[\ell \leq X(u,U) \leq r\right]} \\
&= \frac{\mathbb{P}_{X(u,U)}\left[\ell \leq X(u,U) \leq \eta\right]}{\mathbb{P}_{X(u,U)}\left[\ell \leq X(u,U) \leq r\right]} \\
&= \frac{(p^{\text{new}}(\eta) - \ell p)}{(rp - \ell p)}.
\end{aligned}$$

The probability from case (2) can be shown similarly. Since Line 5 in Algorithm 3.3 updates the search boundaries ℓ and r and their corresponding values ℓp and rp using probabilities (1) and (2), the correctness of the algorithm follows.

For the running time, observe that the number of iterations until the condition of the while loop is no longer satisfied is bounded by $O(\log M)$. Moreover, the running time of one iteration is dominated by the time needed to compute $\mathbf{W}^{\eta}\hat{\mathbf{p}}_0$. Since $\mathbf{W}$ is a $(k+1) \times (k+1)$ dimensional matrix and $\eta \leq M$, Lemma 3.4.10 implies that the matrix powering step can be computed in $O(k^3 \log M)$. Thus, it follows that Algorithm 3.3 can be implemented in $O(k^3 \log^2 M)$ time. □

We next explain how to sample a new distinct vertex, and its corresponding edge of a t-step random walk w, i.e., we discuss component (B). Let $X(u, U)$ be the index computed by BinarySearch routine. We first compute the probability dsitribution $\mathbf{q}$ over vertices in U after performing the next $(X(u, U) - 1)$ steps of the random walk w, conditioning on w not leaving U. Afterwards we proceed to computing the probability distribution $\mathbf{r}$ over the edges leaving U, i.e., edges in the cut $(U, V \setminus U)$, conditioning on $w_0, \ldots, w_t$ and $w_{t+X(u,U)}$ being the first vertex not in U. Formally, for $v \in U$, $z \in V \setminus U$, we have

$$\mathbf{r}(v, z) = \frac{\mathbf{q}(v)\mathbf{w}(v, z)}{R}, \text{ where } R := \sum_{v \in U, z \in V \setminus U} \mathbf{q}(v)\mathbf{w}(v, z). \tag{3.8}$$

Finally, we sample $(w_{t+X(u,U)-1}, w_{t+X(u,U)})$ according to $\mathbf{r}$, where $w_{t+X(u,U)}$ is the first vertex not in U. The lemma below shows that we can efficiently sample from $\mathbf{r}$.

Lemma 3.4.12. *Let w be a t-step random walk and let U with $k = |U|$ be the set of distinct vertices w has visited so far. Given the number of steps $X(u, U)$ needed for w to visit a vertex not in U, there exists an algorithm that samples an edge leaving U, and the first vertex not in U in $O(k^3 \log M)$ time.*

Proof. We start by showing how to compute the distribution $\mathbf{q}$. To this end, recall that $\mathbf{p}_i(v)$ is the probability that $w_{t+i} = v$, conditioned on w not having met any vertex different from U in $w_{t+1}, \ldots, w_{t+i-1}$. Thus, by Equation (3.6), we can use the fast exponentiation routine (Lemma 3.4.10) to compute the vector $\hat{\mathbf{p}}_{X(u,U)-1} = \mathbf{W}^{X(u,U)-1}\hat{\mathbf{p}}_0$. Since by Equation (3.7) we have that $\hat{\mathbf{p}}_{X(u,U)-1}(v) = \mathbf{p}_{X(u,U)-1}(v)$ for each $v \in U$, it follows that $\mathbf{q}(v)$ is exactly $\mathbf{p}_{X(u,U)-1}(v)$. Note that the running time for implementing this step is $O(k^3 \log M)$ as $X(u, U) \leq M$.

We next describe how to efficiently sample from the distribution $\mathbf{r}$. First, it will be helpful to to sample a vertex $v \in U$ conditioning on the $w_{t+X(u,U)}$ being the first vertex not in U. Specifically, we are interested in sampling a vertex $v \in U$ with probability

$$\frac{\mathbf{q}(v) \cdot \mathbf{w}(v, V \setminus U)}{R}, \text{ where } \mathbf{w}(v, V \setminus U) := \sum_{z \in V \setminus U} \mathbf{w}(v, z). \tag{3.9}$$

For being able to efficiently sample from this distribution, we need to compute $\mathbf{w}(v, V \setminus U)$, which in turn may require examining up to $\Omega(n)$ edges incident to v. However, this is not sufficient for our purposes as our ultimate goal is to sample from $\mathbf{r}$ in time only proportional to k. To alleviate this, observe that $\mathbf{w}(v, V \setminus U) = (\mathbf{d}(v) - \sum_{z \in U} \mathbf{w}(v, z))$. Thus, maintaining *weighted* degree $\mathbf{d}_v$ for each $v \in V$, allows us to compute $\mathbf{w}(v, V \setminus U)$ in $O(k)$ time. Similarly, rearranging the sums in the definition of R we get

$$R = \sum_{v \in U,\ z \in V \setminus U} \mathbf{q}(v)\mathbf{w}(v, z) = \sum_{v \in U} \left(\mathbf{q}(v) \cdot \mathbf{w}(v, V \setminus U)\right),$$

which in turn implies that R can be computed in $O(k^2)$ time. The latter gives that the distribution defined in Euqation (3.9) can be computed in $O(k^2)$ time. For sampling a vertex $v \in U$ from this distribution we simply generate a uniformly-random value $x \in [0, 1]$, and then perform binary search on the prefix sum array of the probability distribution. Since computing the prefix sum array and performing binary search can be done in $O(k)$ and $O(\log n)$ time, respectively, we get that sampling $v \in U$ according to distribution defined in Equation (3.9) can be performed in $O(k^2)$ time.

We next explain how to sample an edge (v, z), where $z \in V \setminus U$ and $v \in U$ is the vertex we sampled from above. The probability distribution from which (v, z) is sampled is as follows

$$\frac{\mathbf{w}(v, z)}{\mathbf{w}(v, V \setminus U)}. \tag{3.10}$$

To see the idea behind this choice, note that Equation (3.10) combined with Equation (3.9) yields the distribution $\mathbf{r}$ as defined in Equation (3.8), which ensures that the edge is sampled correctly. However, one complication we face with is that v may be incident to $\Omega(n)$ edges. Remember that for sampling an edge one needs access to the prefix sum array, which is expensive for our purposes. A natural attempt is to compute such an array during preprocessing. Nevertheless, this alone does not suffice as the set U will change over the course of our algorithm. Instead, for every vertex $v \in V$, we maintain an augmented Balanced Binary Tree (BBT) on the edge weights incident to v. Augmented BBT is a data-structure that supports operations such as (1) computing prefix sums and (2) updating the edge weights incident to v, both in $O(\log n)$ time.

We employ the augmented BBT data-structure as follows. First, for each vertex U and the sampled vertex $v \in U$, we update the weights of the edges from v to U to 0 in the augmented BBT of v. We then sample a uniformly-random value $x \in [0, W]$, and use the prefix sums computation in the tree to determine the range in which x lies together with the corresponding edge $(w_{t+X(u,U)-1}, w_{t+X(u,U)})$, where $w_{t+X(u,U)}$ is the first vertex not in U. After having sampled the edge, we undo all the changes we performed in the augmented BBT of v. It follows that sampling an edge according to Equation (3.10) can be implemented in $O(k \log n)$. Putting together the above running times, we conclude that sampling an edge leaving U as well as the first vertex not in U can be implemented in $O(k^3 \log M)$ time. □

The last ingredient we need is an efficient way to sample the sum of weights in the random walk starting at w_t and ending at $w_{t+X(u,U)}$, where $X(u, U)$ is the number of steps needed for the walk to leave the vertex set U (Component (C)). In other words, we need to sample the following sum

$$\sum_{i=t+1}^{t+X(u,U)} \frac{1}{\mathbf{w}(w_{i-1}, w_i)}.$$

We accomplish this task by employing a doubling technique. To illustrate, for any pair of vertices $u, v \in V$ and $s(w)$ as defined in Equation (3.3), let

$$f^{u,v}_{s(w),\ell} \tag{3.11}$$

be the probability mass function of $s(w)$ conditioning on (1) w being a random walk that starts at u and ends at v, i.e., $w \sim w_{u,v}$ and (2) length of the walk $\ell(w)$ is ℓ in G. Then it can be shown that

$$f^{u,v}_{s(w),\ell} = \sum_{y \in V} \left(f^{u,y}_{s(w),\ell/2} * f^{y,v}_{s(w),\ell/2} \right),$$

where $*$ denotes the convolution between two probability mass functions. Equivalently, the convolution is the probability mass function of the sum of the two corresponding random variables. The above relation suggests that if (1) we have some *approximate* representation of the probability mass functions $f^{u,v}_{s(w),\ell/2}$ for all $u, v \in V$, and (2) we are able to compute the convolution of the two mass functions under such representation, we can produce approximations for $f^{u,v}_{s(w),\ell}$, where $u, v \in V$. This idea is formalized in the following lemma.

Lemma 3.4.13. *Let $G = (V, E, \mathbf{w})$ be a undirected, weighted graph with $\mathbf{w}(e) = [1, n^c]$ for each $e \in E$, where c is a positive constant. For any finite random walk w of length ℓ with $\ell \leq n^d$, where d is a positive constant, let $s(w)$ be the sum of the inverse of its edge weights, i.e.,*

$$s(w) = \sum_{i=1}^{\ell} \frac{1}{\mathbf{w}(w_{i-1}, w_i)}.$$

Moreover, for any $u, v \in V$, let

$$f^{u,v}_{s(w),\ell}$$

be the probability mass function of $s(w)$ conditioning on (1) w being a random walk that starts at u and ends at v, and (2) length of the walk $\ell(w)$ is ℓ in G. Then, for any pair $u, v \in V$, there exists an algorithm that that samples from $f^{u,v}_{s(w),\ell}$ and outputs a sampled $s(w)$ up to a $(1+\epsilon)$ multiplicative error in $\tilde{O}(n^3\epsilon^{-2})$ time.

We finally describe a procedure that generates a β-shorted walk with respect to some vertex v and set of terminals K. Concretely, the algorithm maintains (1) a set U, initialized to $\{v\}$, of the distinct vertices visited so far by a random walk w starting at v, (2) the number of steps t the walk w has performed so far and (3) two lists L_w and L_s, initially set to empty, containing the first occurrences of distinct vertices of w and the weightes of the sub-walks induced by the distinct vertices, respectively. Next, as long as w does not hit a vertex in K or there are vertices in the component containing v that are still not visited by w, for the next $\Theta(\beta^{-1} \log n)$ steps, the algorithm repeatedly generates a new vertex not in the current U by using components (A), (B) and (C). In each iteration, the maintained quantities U, t, L_w and L_s are updated accordingly. Note that this procedure indeed outputs all

Algorithm 3.4: GENERATESINGLEWALK(G, K, v)

Input : Weighted graph $G = (V, E, \mathbf{w})$ with $\mathbf{w}(e) = [1, n^c]$ for each $e \in E$ and $c > 0$, a set of vertices $K \subseteq V$, a vertex $v \in V$ such that the component containing v contains at least one vertex in K

Output : Two lists L_w and L_s containing the first occurrences of distinct vertices of a random walk w starting at v and the weights of the sub-walks induced by the distinct vertices, respectively

1 Set $U \leftarrow \{v\}$, $k \leftarrow |U|$, and let $u \leftarrow v$ be the current vertex
2 Let $t \leftarrow 0$ be the index of current step of random walk w, i.e., $w_t = u$
3 Let L_w and L_s be two lists, initially set to empty
4 **for** *each* $i = 1, \ldots, \Theta(\beta^{-1} \log n)$ **do**
5 Let $\mathbf{W}$ be a matrix of dimension $(k+1) \times (k+1)$ as defined in Equation (3.6)
6 Set $\hat{\mathbf{p}}_0 = [\mathbf{p}_0 \quad 0]^\top$, where $\mathbf{p}_0(u) \leftarrow 1$, and $\mathbf{p}_0(\hat{u}) \leftarrow 0$ for every $\hat{u} \in U \setminus u$
7 Set $X(u, U) \leftarrow$ BINARYSEARCH$(\mathbf{W}, \hat{\mathbf{p}}_0, O(m^3))$
8 Compute the probability distribution $\mathbf{q}$ over vertices in U after $(X(u, U) - 1)$ steps of the random walk w, conditioning on w not leaving U
9 Compute the probability distribution $\mathbb{R}$ over the the edges in $(U, V \setminus U)$ conditioning on $w_0, \ldots, w_t$ and $w_{t+X(u,U)}$ being the first vertex not in U. Concretely, for $v \in U$, $z \in V \setminus U$,

$$\mathbf{r}(v, z) = \frac{\mathbf{q}(v)\mathbf{w}(v, z)}{R}, \text{ where } R \leftarrow \left(\sum_{v \in U,\ z \in V \setminus U} \mathbf{q}(v)\mathbf{w}(v, z) \right).$$

10 Sample $(w_{t+X(u,U)-1}, w_{t+X(u,U)})$ according to $\mathbf{r}(w_{t+X(u,U)-1}, w_{t+X(u,U)})$
11 Set $e^{\text{new}} \leftarrow (w_{t+X(u,U)-1}, w_{t+X(u,U)})$
12 Invoke Lemma 3.4.13 in the inducted graph $G[U]$ to sample
13

$$s = \sum_{j=t+1}^{t+X(u,U)-1} \frac{1}{\mathbf{w}(w_{j-1}, w_j)}.$$

14 Append $w_{t+X(u,U)}$ to L_w and $(s + 1/\mathbf{w}(e^{\text{new}}))$ to L_s
15 **if** $w_{t+X(u,U)} \in K$ **then**
16 Go to Line 20
17 **else**
18 Set $t \leftarrow (t + X(u, U))$, $u \leftarrow w_{t+X(u,U)}$, $U \leftarrow U \cup \{u\}$, and $k \leftarrow (k+1)$
19 If U covers the entire component, go to Line 20. Otherwise, $i \leftarrow (i+1)$
20 **return** lists L_w and L_s.

necessary information we need from a β-shorted walk. A detailed implementation of the algorithm is summarized in Figure 3.4.

We now have all the necessary tools to prove Lemma 3.4.9.

Proof of Lemma 3.4.9. We first show correctness. By Lemma 3.4.11 it follows that BINARYSEARCH correctly samples the number of steeps before a walk meets a new vertex. Next, Lemma 3.4.12 implies that the we can sample the new dis-

tinct vertex and its corresponding edge. Finally, by Lemma 3.4.13 we know that the weight of each sub-walk of a β-shorted walk is approximated within a $(1+\epsilon)$ multiplicative error. Bringing these approximation together we get that the weight of the β-shorted walk itself is approximated within the same multiplicative error.

We now analyse the running time of procedure GenerateSingleWalk. We start by bounding the cover time of G, which in turn bounds the number of steps for a random walk to meet a new vertex. To this end, note that it takes expected $O(m^2)$ time to meet a vertex in the same component ([16]). Thus, if we perform a random walk of length $O(m^3)$ we are guaranteed that it covers ever vertex in the component containing the starting vertex, with high probability.

Next, we analyze the running time for the steps executed within one iteration of the for loop. Observe that $k = |U| = O(\beta^{-1}\log n)$ at any point of time throughout our algorithm. The latter together with Lemma 3.4.11 give that it takes $O(k^3 \log^2 M) = \tilde{O}(\beta^{-3})$ time to sample the minimum number of steps for a random walk to visit a vertex not in U, where $M = O(m^3)$ by the discussion above. Furthermore, by Lemma 3.4.12 we can sample the new vertex not in U, and its corresponding edge in $\tilde{O}(\beta^{-3})$ time. Finally, Lemma 3.4.13 implies that the weight $s(w)$ of the random sub-walk between the current vertex and the new generated vertex can be approximately sampled in $\tilde{O}(\beta^{-3}\epsilon^{-2})$ time. The latter holds because Lemma 3.4.13 is invoked on top of the graph $G[U]$ for which $|V(G[U])| = O(\beta^{-1}\log n)$. Combining the above running times, we get that one iteration can be implemented in $\tilde{O}(\beta^{-3}\epsilon^{-2})$ time. Since there are $O(\beta^{-1}\log n)$ iterations, we conclude that the overall running time of our procedure is $\tilde{O}(\beta^{-4}\epsilon^{-2})$. □

We now present the procedure for generating a Schur complement on weighted graphs. The idea behind this algorithm is the same as in the unweighted setting, except that now we use GenerateSingleWalk to extract the information needed to simulate β-shorted walks. For the sake of completeness we summarize the details of this modified procedure in Algorithm 11.

Lemma 3.4.14. *Algorithm 11 runs in $\tilde{O}(m\beta^{-4}\epsilon^{-4})$ time and outputs a graph H satisfying $\mathbf{L}_H \approx_\epsilon \mathbf{SC}(G, K)$, with high probability.*

Proof. We first bound the running time of Algorithm 11. By Lemma 3.4.9, the time needed to generate a β-shorted walk is $\tilde{O}(\beta^{-4}\epsilon^{-2})$. Combining this with the fact that the algorithm generates $\rho m = \tilde{O}(m\epsilon^{-2})$ walks, it follows that the running time of the algorithms is dominated by $\tilde{O}(m\beta^{-4}\epsilon^{-4})$.

We next show the correctness of our procedure. First, note that procedure GenerateSingleWalk generates a valid β-shorted walk with its weight being approximated up to a $(1+\epsilon)$ multiplicative error (Lemma 3.4.9). Assume for now that there is an oracle that fixes this approximate weight of a walk back to its original exact weight. Then the collection of generated walks from Algorithm 11 forms the set W of β-shorted walks, and let $\hat{H}$ be the corresponding output graph. By Theorem 3.4.3, with high probability, each of the walks that starts at a component containing a ver-

Algorithm 3.5: InitializeWeighted(G, K', β)

Input : Weighted graph $G = (V, E, \mathbf{w})$ with $\mathbf{w}(e) = [1, n^c]$ for each $e \in E$ and $c > 0$, set of vertices $K' \subseteq V$ such that $|K'| \leq O(\beta m)$, and $\beta \in (0, 1)$

Output : Approximate Schur Complement H and union of β-shorted walks W

Set $K \leftarrow K'$, $H \leftarrow (V, \emptyset)$ and $W \leftarrow \emptyset$

For each edge $e = (u, v)$ in G, let $K \leftarrow K \cup \{u, v\}$ with probability β

Let $\rho \leftarrow O(\log n \epsilon^{-2})$ be the sampling overhead according to Theorem 3.3.1

for *each edge* $e = (u, v) \in E$ *and each* $i = 1, \ldots, \rho$ **do**

  Using Algorithm 3.4, generate a random walk $w_1(e, i)$ from u until $\Theta(\beta^{-1} \log n)$ different vertices have been hit, it reaches K, or it has hit every edge in its component

  Using Algorithm 3.4, generate a random walk $w_2(e, i)$ from v until $\Theta(\beta^{-1} \log n)$ different vertices have been hit, it reaches K, or it hast hit every edge in its component

  if *both walks reach* K *at* t_1 *and* t_2 *respectively* **then**

    Connect $w_1(e, i)$, e and $w_2(e, i)$ to form a walk $w(e, i)$ between t_1 and t_2

    Let $s \leftarrow s(w_1(e, i)) + s(w_2(e_i)) + 1/\mathbf{w}(e)$

    Add an edge (t_1, t_2) with weight $1/(\rho s)$ to H

    Add $w(e, i)$ to W

return H and W

tex in K hits K. Conditioning on the latter, Theorem 3.3.1 gives that with high probability, $\mathbf{L}_{\hat{H}} \approx_\epsilon \mathbf{SC}(G, K)$.

Finally, let H be the graph where the edge weights are correct up to a $(1 + \epsilon)$ multiplicative error. In other words, the weight of each edge e in H differs from the corresponding weight $\mathbf{w}_{\hat{H}}(e)$ in $\hat{H}$ by $\epsilon \mathbf{w}_{\hat{H}}(e)$. Summing over all the edges we get that $\mathbf{L}_H \approx_\epsilon \mathbf{L}_{\hat{H}}$. Since $\mathbf{L}_{\hat{H}} \approx_\varepsilon \mathbf{SC}(G, K)$ by the discussion above, we get that $\mathbf{L}_H \approx_{O(\epsilon)} \mathbf{SC}(G, K)$. Scaling ϵ appropriately completes the correctness. □

We now have all the necessary tools to present our dynamic algorithm for maintaining the collection of walks W (equivalently, the approximate Schur complement H), on weighted graphs.

Proof of Lemma 3.4.8. Similarly to the unweighted case, we give a two-level data-structure for dynamically maintaining Schur complements on weighted graphs. Specifically, we keep the terminal set K of size $\Theta(m\beta)$. This entails maintaining

1. an approximate Schur complement H of G with respect to K (Theorem 3.3.1),
2. a dynamic spectral sparsifier $\tilde{H}$ of H (Lemma 3.2.4).

We implement the procedure Initialize by running Algorithm 11, which produces a graph H and then compute a spectral sparsifier $\tilde{H}$ of H using Lemma 3.2.4. Note that by construction of our data-structure, every update in H will be handled by the black-box dynamic sparsifier $\tilde{H}$.

Similarly to the unweighted case, operations INSERT and DELETE are reduced to adding terminals to the set K and we refer the reader to the previous section for details on this reduction. Thus, the bulk of our effort is devoted to implementing the procedure ADDTERMINAL. Let u be a non-terminal vertex that we want to append to K. We set $K \leftarrow K \cup \{u\}$, and then shorten all the walks at the first location they meet u. This shortening of walks induces in turn edge insertions and deletions to H, which are then processed by $\tilde{H}$. To quickly locate the first appearances of u in the random walks from W, we maintain a linked list W_u for each $u \in V$. This linked list contains the first appearances of w in the collections of random walks W. Note that constructing such lists can be performed at no additional cost during preprocessing phase, since Algorithm 3.4 directly gives the first appearances of vertices in every walk belonging to W. After locating the first appearances of u, we cut the walks in these locations, delete the corresponding affected walks (together with their weight from H), and insert the new shorter walks to H. Note that we can simply use arrays to represent each random walk in W.

We next analyze the performance of our data-structure. Let us start with the preprocessing time. First, by Lemma 3.4.14 we get that the cost for constructing H on a graph with m edges is bounded by $\tilde{O}(m\beta^{-4}\epsilon^{-4})$. Next, since H has $\tilde{O}(m\epsilon^{-2})$ edges, constructing $\tilde{H}$ takes $\tilde{O}(m\epsilon^{-4})$ time. Thus, the amortized time of INITIALIZE operation is bounded by $\tilde{O}(m\beta^{-4}\epsilon^{-4})$.

We now analyze the update operations. By the above discussion, note that it suffices to bound the time for adding a vertex to K, which in turn (asymptotically) bounds the update time for edge insertions and deletions. The main observation we make is that adding a vertex to K only shortens the existing walks, and by the above discussion we can find such walks in time proportional to the amount of edges deleted from the walk. Since the walk needed to be generated in the INITIALIZE operation, the deletion of these edges take equivalent time to generating them. Moreover, we note that (1) handling the updates in $\tilde{H}$ induced by H introduces additional $O(\mathrm{poly}(\log n)\epsilon^{-2})$ overheads, and (2) adding or deleting ρ edges until the next rebuild costs $\tilde{O}(\beta m\epsilon^{-2})$, since we process only up to βm operations. These together imply that the amortized cost for adding a terminal can be charged against the preprocessing time, which is bounded by $\tilde{O}(m\beta^{-4}\epsilon^{-4})$, up to poly-logarithmic factors. Thus it follows that the operations ADDTERMINAL, INSERT and DELETE can be implemented in $\tilde{O}(1)$ amortized update time. □

3.4.4 Dynamic All-Pair Effective Resistance on Weighted Graphs

Following exactly the same arguments as in the proof of Theorem 3.1.3, we can use the above data-structure to efficiently maintain effective resistances on weighted, undirected dynamic graphs.

Theorem 3.4.15. *For any given error threshold $\epsilon > 0$, there is a data-structure for maintaining a weighted, undirected multi-graph $G = (V, E, \mathbf{w})$ with up to m edges that supports the following operations in $\tilde{O}(m^{5/6}\epsilon^{-4})$ expected amortized time:*

- *INSERT(u, v, w): Insert the edge (u, v) with resistance $1/w$ in G.*
- *DELETE(u, v): Delete the edge (u, v) from G.*
- *EFFECTIVERESISTANCE(s, t): Return a $(1 \pm \epsilon)$-approximation to the effective resistance between s and K in the current graph G.*

Proof. Let $\mathcal{D}(\tilde{H})$ denote the data structure that maintains a dynamic (sparse) Schur complement $\tilde{H}$ of G (Lemma 3.4.8). Since $\mathcal{D}(\tilde{H})$ supports only up to βm operations, we rebuild $\mathcal{D}(\tilde{H})$ on the current graph G after such many operations. Note that the operations INSERT and DELETE on G are simply passed to $\mathcal{D}(\tilde{H})$. For processing the query operation EFFECTIVERESISTANCE(s, t), we declare s and t terminals (using the operation ADDTERMINAL of $\mathcal{D}(\tilde{H})$), which ensures that they are both now contained in $\tilde{H}$. Finally, we compute the (approximate) effective resistance between s and t in $\tilde{H}$ using Lemma 3.2.1.

We now analyze the performance of our data-structure. Recall that the insertion or deletion of an edge in G can be supported in $\tilde{O}(1)$ expected amortized time by $\mathcal{D}(\tilde{H})$. Since our data-structure is rebuilt every βm operations, and rebuilding $\mathcal{D}(\tilde{H})$ can be implemented in $\tilde{O}(m\beta^{-4}\epsilon^{-4})$ time, it follows that the amortized cost per edge insertion or deletion is

$$\frac{\tilde{O}(m\beta^{-4}\epsilon^{-4})}{\beta m} = \tilde{O}(\beta^{-5}\epsilon^{-4}).$$

The cost of any (s, t) query is dominated by (1) the cost of declaring s and t terminals and (2) the cost of computing the (s, t) effective resistance to ϵ accuracy on the graph $\tilde{H}$. Since (1) can be performed in $\tilde{O}(1)$ time, we only need to analyze (2). We do so by first giving a bound on the size of K. To this end, note that each of the m edges in the current graph adds two vertices to K with probability β independently. By a Chernoff bound, the number of random augmentations added to K is at most $2\beta m$ with high probability. In addition, since $\mathcal{D}(\tilde{H})$ is rebuilt every βm operations, the size of K never exceeds $4\beta m$ with high probability. The latter also bounds the size of $\tilde{H}$ by $\tilde{O}(\beta m\epsilon^{-2})$ and gives that the query cost is $\tilde{O}(\beta m\epsilon^{-2})$.

Combining the above bounds on the update and query time, we obtain the following trade-off

$$\tilde{O}\left((\beta m + \beta^{-5})\epsilon^{-4}\right),$$

which is minimized when $\beta = m^{-1/6}$, thus giving an expected amortized update and query time of

$$\tilde{O}(m^{5/6}\epsilon^{-4}).$$ □

3.5 Dynamic Laplacian Solver in Sub-linear Time

In this section we extend our dynamic approximate Schur complement algorithm to obtain a dynamic Laplacian solver for unweighted, bounded degree graphs. Specifically, as described in Theorem 3.1.2, our goal is to design a data-structure that

maintains a solution to the Laplacian system $\mathbf{Lx} = \mathbf{b}$ under updates to both the underlying graph and the demand vector vector $\mathbf{b}$ while being able to query a few entries of the solution vector. For the sake of exposition, in what follows we assume that the underlying graph is always connected.

Consider the dynamic Schur complement data-structure provided by Lemma 3.1.1. If the demand vector $\mathbf{b}$ has up to $O(\beta m)$ non-zero entries, for some parameter $\beta \in (0, 1)$, we can simply incorporate the vertices corresponding to these entries in the terminal set K using operation AddTerminal of the dynamic Schur complement data-structure (Lemma 3.1.1). Upon receipt of a query index, we add the corresponding vertex to the Schur complement and (approximately) solve a Laplacian system on the maintained Schur complement. The obtained solution vector can then be lifted back to the Laplacian matrix using the following lemma, which we introduced in the preliminaries.

Lemma 3.5.1 (Restatement of Lemma 3.2.6)**.** *Let $\mathbf{x}_K$ be a solution vector such that $\mathbf{SC}(G, K)\mathbf{x}_K = \mathbf{P}(K)\mathbf{b}$. Then there exists an extension $\mathbf{x}$ of $\mathbf{x}_K$ such that $\mathbf{Lx} = \mathbf{b}$.*

As we argued in Section 3.3, this approach leads to a dynamic Laplacian solver with $O(m^{3/4})$ amortized update time per operation. Moreover, note that the algorithm applies to any undirected, unweighted graph. However, the prime difficulty for constructing a dynamic solver is in handling the case where $\mathbf{b}$ has a large number of non-zero entries, i.e., $\|\mathbf{b}\|_0 = \Omega(n)$, thus preventing us from obtaining a sub-linear algorithm using the reduction above. We alleviate this by projection this demand vector onto the current set of terminals and showing that such a projection can be maintained dynamically while introducing controllable error in the approximation guarantee. At a high level, our solver can be viewed as an one layer version of sparsified block-Cholesky algorithms [174].

We next discuss specific implementation details. Recall that $\mathbf{P}(K)$ is the matrix projection of non-terminal vertices F onto K. By Lemma 3.2.6, it is sufficient to maintain a solution $\mathbf{x}_T = \mathbf{SC}(G, T)^\dagger \mathbf{P}(T)\mathbf{b}$ dynamically. Since Lemma 3.1.1 already allows us to maintain $\mathbf{SC}G, K$, we need to devise a routine that maintains the projection $\mathbf{P}(T)\mathbf{b}$ of $\mathbf{b}$ under vertex additions to the terminal set.

To this end, we describe an algorithm that maintains such a projection which in turn allows us to again achieve sub-linear running times. The algorithm itself can be viewed as a numerically minded generalization of the approach for the small-support case. Let S denote the current set of terminals that the algorithm maintains (S and K will always be equal, and we differentiate between them only for the sake of presentation). We initialize S with $O(\beta m)$ vertices from the corresponding entries in $\mathbf{b}$ that have the largest value. Our key structural observation is that if the entries of $\mathbf{b}$ are small, adding vertices to S does not change the projection significantly. To measure the error incurred by declaring some vertex a terminal, we exploit the fact that the projection $\mathbf{P}(S)\mathbf{b}$ itself is tightly connected to specific random walks in the underlying graph. We then argue that it is possible to reuse earlier projections, even when new terminals are added to S, while paying an error

corresponding to the lengths of these random walks and the magnitude of entries in $\mathbf{b}$. Finally, we analyze how to control the accumulation of these errors over a sequence of terminal additions, and also describe an initialization procedure that involves solving a Laplacian system for computing the starting (approximate) projection vector. These together lead to the main lemma of this section, whose implementation details and analysis are deferred to Subsection 3.5.1.

Lemma 3.5.2. *Given an error parameter $\epsilon > 0$, an unweighted unweighted bounded-degree $G = (V, E)$ with n vertices, a vector $\mathbf{b} \in \mathbb{R}^n$ in the image of $\mathbf{L}$, a subset of terminal vertices S' and a parameter $\beta \in (0, 1)$ such that $|S'| = O(\beta m)$, there is a data-structure DynamicProj(G, S', β) for maintaining a vector $\tilde{\mathbf{b}}$ with $\left\|\tilde{\mathbf{b}} - \mathbf{P}(S)\mathbf{b}\right\|_{\mathbf{L}^\dagger} \leq \epsilon \|\mathbf{b}\|_{\mathbf{L}^\dagger}$ for some S with $S' \subseteq S$, $|S| = O(\beta m)$, while supporting at most $\beta^3 m^{1/2}\epsilon(\text{poly}\log n)^{-1}$ operations in the following running times:*

- *Initialize(G, S', β): Initialize the data-structure in $\tilde{O}(m)$ time.*
- *Insert(u, v): Insert the edge (u, v) to G in $O(1)$ time while keeping G bounded-degree.*
- *Delete(u, v): Delete the edge (u, v) from G in $O(1)$ time.*
- *Change$(u, \mathbf{b}'(u), v, \mathbf{b}'(v))$: Change $\mathbf{b}(u)$ to $\mathbf{b}'(u)$ and changes $\mathbf{b}(v)$ to $\mathbf{b}'(v)$ while keeping $\mathbf{b}$ in the range of $\mathbf{L}$ in $O(1)$ time.*
- *AddTerminal(u): Add u to S in $O(1)$ time.*
- *Query$()$: Output the maintained $\tilde{\mathbf{b}}$ in $O(\beta m)$ time.*

The following lemma, whose proof will be shortly provided, allows us to combine the approximation guarantees of the data-structures (1) dynamic Schur complement and (2) dynamic Projection.

Lemma 3.5.3. *Let $0 < \epsilon \leq \frac{1}{2}$. Let k be a positive number such that $\|\mathbf{b}\|_{\mathbf{L}^\dagger} \leq k$. Suppose $\tilde{\mathbf{L}} \approx_\epsilon \mathbf{L}$, $\left\|\tilde{\mathbf{b}} - \mathbf{b}\right\|_{\mathbf{L}^\dagger} \leq \epsilon k$ and $\left\|\tilde{\mathbf{x}} - \tilde{\mathbf{L}}^\dagger\tilde{\mathbf{b}}\right\|_{\tilde{\mathbf{L}}} \leq \epsilon \left\|\tilde{\mathbf{L}}^\dagger\tilde{\mathbf{b}}\right\|_{\tilde{\mathbf{L}}}$. Then $\left\|\tilde{\mathbf{x}} - \mathbf{L}^\dagger\mathbf{b}\right\|_{\mathbf{L}} \leq 10\epsilon k$.*

We now have all the necessary tools to present the data-structure for solving Laplacian systems in bounded-degree graphs, which essentially entails combining Lemma 3.1.1 and Lemma 3.5.2.

Proof of Theorem 3.1.2. Let $\mathcal{D}(\tilde{H})$ and $\mathcal{D}(\tilde{\mathbf{b}})$ denote the data-structure that maintains a dynamic (sparse) Schur complement $\tilde{H}$ of G (Lemma 3.1.1) and an approximate dynamic Projection $\tilde{\mathbf{b}}$ of $\mathbf{P}(S)\mathbf{b}$ (Lemma 3.5.2), respectively. Set $\epsilon \leftarrow (\epsilon/10)$ for both data-structures. Our dynamic solver simultaneously maintains $\mathcal{D}(\tilde{H})$ and $\mathcal{D}(\tilde{\mathbf{b}})$. Since $\mathcal{D}(\tilde{\mathbf{b}})$ supports only up to $\beta^3 m^{1/2}\epsilon(\text{poly}\log n)^{-1}$, we rebuild both data-structures after such many operations.

We now describe the implementation of the operations. First, we find the first βm entries with maximum value in $\mathbf{b}$. We then take the corresponding vertices

and initialize S' and K' to be these βm vertices. The implementation of these data-structures involves including the endpoints of each edge with probability β to S and K, respectively. We couple these algorithms such that $S = K$, and this property will be maintained throughout the algorithm. The operations INSERT and Delete on G are simply passed to $\mathcal{D}(\tilde{H})$ and $\mathcal{D}(\tilde{\mathbf{b}})$. The operation CHANGE$(u, \mathbf{b}'(u), v, \mathbf{b}'(v))$ is passed to $\mathcal{D}(\tilde{\mathbf{b}})$. Upon receipt of a query $\mathbf{x}(u)$, for some vertex $u \in V$, i.e., operation SOLVE(u), we declare u a terminal (using the operation ADDTERMINAL(u) of both $\mathcal{D}(\tilde{H})$ and $\mathcal{D}(\tilde{\mathbf{b}})$). We then proceed by extracting an approximate Schur complement $\tilde{H}$ of G from $\mathcal{D}(\tilde{H})$ and an approximate projection vector $\tilde{\mathbf{b}}$ from $\mathcal{D}(\tilde{\mathbf{b}})$. Finally, using a black-box Laplacian solver [168], we compute a solution vector $\tilde{\mathbf{x}}_K$ to the system $\mathbf{L}_{\tilde{H}}\tilde{\mathbf{x}}_K = \tilde{\mathbf{b}}$ and output $\tilde{\mathbf{x}}_K(u)$ (this is possible since u was added to K).

We next show the correctness of the operation SOLVE(u). The Laplacian solver guarantees that the vector $\tilde{\mathbf{x}}_K$ satisfies

$$\left\|\tilde{\mathbf{x}}_K - \mathbf{L}_{\tilde{H}}^{\dagger}\tilde{\mathbf{b}}\right\|_{\mathbf{L}_{\tilde{H}}} \leq (\epsilon/10)\left\|\mathbf{L}_{\tilde{H}}^{\dagger}\tilde{\mathbf{b}}\right\|_{\mathbf{L}_{\tilde{H}}}. \tag{3.12}$$

Data-structure $\mathcal{D}(\tilde{\mathbf{b}})$ guarantees that

$$\left\|\tilde{\mathbf{b}} - \mathbf{P}(K)\mathbf{b}\right\|_{\mathbf{SC}(G,T)^{\dagger}} \leq (\epsilon/10)\left\|\mathbf{b}\right\|_{\mathbf{L}^{\dagger}}. \tag{3.13}$$

Note that $\|\mathbf{P}(K)\mathbf{b}\|_{\mathbf{SC}(G,K)^{\dagger}} \leq \|\mathbf{b}\|_{\mathbf{L}^{\dagger}}$. Bringing together Equations (3.12) and (3.13) and applying Lemma 3.5.3 with $k = \|\mathbf{b}\|_{\mathbf{L}^{\dagger}}$, $\mathbf{L} := \mathbf{SC}(G, K)$, $\mathbf{b} := \mathbf{P}(K)\mathbf{b}$, $\mathbf{L}_{\tilde{H}}$ and $\tilde{\mathbf{b}}$ yield

$$\left\|\tilde{\mathbf{x}}_K - \mathbf{SC}(G,K)^{\dagger}\mathbf{P}(K)\mathbf{b}\right\|_{\mathbf{SC}(G,K)} \leq \epsilon k.$$

Using Lemma 3.2.6 we can lift the vector $\tilde{\mathbf{x}}_K$ to a solution $\tilde{\mathbf{x}}$ such that

$$\left\|\tilde{\mathbf{x}} - \mathbf{L}^{\dagger}\mathbf{b}\right\|_{\mathbf{L}} \leq \epsilon k = \epsilon\left\|\mathbf{b}\right\|_{\mathbf{L}^{\dagger}} = \epsilon\left\|\mathbf{L}^{\dagger}\mathbf{b}\right\|_{\mathbf{L}}.$$

Finally, we bound the running time of our dynamic solver. Changes in the demand vector $\mathbf{b}$ can be performed in $O(1)$ times, thus having negligible affect in our running times. The insertion or deletion of an edge in G can be supported in $\tilde{O}(1)$ expected amortized time by both $\mathcal{D}(\tilde{H})$ and $\mathcal{D}(\tilde{\mathbf{b}})$. Since we build our data-structures every $\beta^3 m^{1/2}\epsilon(\text{poly}\log n)^{-1}$ operations, and the total rebuild cost is dominated by $\tilde{O}(m\beta^{-2}\epsilon^{-4})$, it follows that the amortized cost per edge insertion or deletion is

$$\frac{\tilde{O}(m\beta^{-2}\epsilon^{-4})}{\beta^3 m^{1/2}\epsilon(\text{poly}\log n)^{-1}} = \tilde{O}(m^{1/2}\beta^{-5}\epsilon^{-5}).$$

The cost of any query is dominated by (1) the cost of declaring the queried vertex u a terminal and (2) the cost of extracting $\tilde{H}$ and $\tilde{\mathbf{b}}$. Since (1) can be performed in $\tilde{O}(1)$ amortized time, we only need to analyze (2). Size of the terminal set $S = K$,

which can be easily shown to be $O(\beta m)$ with high probability, immediately implies that the running time for (2) is dominated by $\tilde{O}(\beta m\epsilon^{-2}) = \tilde{O}(\beta m\epsilon^{-5})$, which in turn bounds the query cost.

Combining the above bounds on the query and update time, we obtain the following trade-off

$$\tilde{O}\left((m^{1/2}\beta^{-5} + \beta m)\epsilon^{-5}\right)$$

which is minimized when $\beta = m^{-1/12}$, thus giving an expected amortized update and query time of

$$\tilde{O}(m^{11/12}\epsilon^{-5}).$$

We can replace m by n in the above running time guarantee since by our assumption, G has bounded-degree throughout the algorithm. □

We next prove Lemma 3.5.3.

Proof of Lemma 3.5.3. We will use triangle inequality to decompose the error as:

$$\begin{aligned}\left\|\tilde{\mathbf{x}} - \mathbf{L}^{\dagger}\mathbf{b}\right\|_{\mathbf{L}} &= \left\|\tilde{\mathbf{x}} - \tilde{\mathbf{L}}^{\dagger}\tilde{\mathbf{b}} + \tilde{\mathbf{L}}^{\dagger}\tilde{\mathbf{b}} - \tilde{\mathbf{L}}^{\dagger}\mathbf{b} + \tilde{\mathbf{L}}^{\dagger}\mathbf{b} - \mathbf{L}^{\dagger}\mathbf{b}\right\|_{\mathbf{L}} \\ &\leq \left\|\tilde{\mathbf{x}} - \tilde{\mathbf{L}}^{\dagger}\tilde{\mathbf{b}}\right\|_{\mathbf{L}} + \left\|\tilde{\mathbf{L}}^{\dagger}\tilde{\mathbf{b}} - \tilde{\mathbf{L}}^{\dagger}\mathbf{b}\right\|_{\mathbf{L}} + \left\|\tilde{\mathbf{L}}^{\dagger}\mathbf{b} - \mathbf{L}^{\dagger}\mathbf{b}\right\|_{\mathbf{L}}, \qquad (3.14)\end{aligned}$$

and bound each of them separately.

1. The first term can be bounded by first invoking the similarity of $\mathbf{L}$ and $\tilde{\mathbf{L}}$ to change the norm to $\tilde{\mathbf{L}}$, and applying the guarantees of the solve involving $\tilde{\mathbf{L}}$:

$$\left\|\tilde{\mathbf{x}} - \tilde{\mathbf{L}}^{\dagger}\tilde{\mathbf{b}}\right\|_{\mathbf{L}} \leq \sqrt{(1+2\epsilon)}\left\|\tilde{\mathbf{x}} - \tilde{\mathbf{L}}^{\dagger}\tilde{\mathbf{b}}\right\|_{\tilde{\mathbf{L}}} \leq 2\left\|\tilde{\mathbf{x}} - \tilde{\mathbf{L}}^{\dagger}\tilde{\mathbf{b}}\right\|_{\tilde{\mathbf{L}}} \leq 2\epsilon\left\|\tilde{\mathbf{b}}\right\|_{\tilde{\mathbf{L}}^{\dagger}}.$$

   This norm can in turn be transferred back to $\mathbf{L}$, and the discrepancy between $\mathbf{b}$ and $\tilde{\mathbf{b}}$ absorbed using triangle inequality:

$$\leq 3\epsilon\left\|\tilde{\mathbf{b}}\right\|_{\mathbf{L}^{\dagger}} \leq 3\epsilon\left(\|\mathbf{b}\|_{\mathbf{L}^{\dagger}} + \left\|\tilde{\mathbf{b}} - \mathbf{b}\right\|_{\mathbf{L}^{\dagger}}\right) \leq 3\epsilon(1+\epsilon)k \leq 5\epsilon k.$$

2. The second term follows from combining the norms in $\tilde{\mathbf{L}}$ and $\mathbf{L}$ using the approximations between these matrices:

$$\left\|\tilde{\mathbf{L}}^{\dagger}\tilde{\mathbf{b}} - \tilde{\mathbf{L}}^{\dagger}\mathbf{b}\right\|_{\mathbf{L}} = \left\|\tilde{\mathbf{b}} - \mathbf{b}\right\|_{\tilde{\mathbf{L}}^{\dagger}\mathbf{L}\tilde{\mathbf{L}}^{\dagger}} \leq 2\left\|\tilde{\mathbf{b}} - \mathbf{b}\right\|_{\tilde{\mathbf{L}}^{\dagger}\tilde{\mathbf{L}}\tilde{\mathbf{L}}^{\dagger}} = 2\left\|\tilde{\mathbf{b}} - \mathbf{b}\right\|_{\tilde{\mathbf{L}}^{\dagger}},$$

   and once again converting the norm back from $\tilde{\mathbf{L}}$ to $\mathbf{L}$:

$$\leq 4\left\|\tilde{\mathbf{b}} - \mathbf{b}\right\|_{\mathbf{L}^{\dagger}} \leq 4\epsilon k.$$

3. The third term can first be written in terms of the norm of $\mathbf{b}$ against a matrix involving the difference between $\mathbf{L}$ and $\tilde{\mathbf{L}}$:

$$\left\|\tilde{\mathbf{L}}^{\dagger}\mathbf{b}-\mathbf{L}^{\dagger}\mathbf{b}\right\|_{\mathbf{L}}=\left\|\left(\tilde{\mathbf{L}}^{\dagger}-\mathbf{L}^{\dagger}\right)\mathbf{b}\right\|_{\mathbf{L}}=\left\|\mathbf{L}^{\dagger/2}\mathbf{b}\right\|_{\left(\mathbf{L}^{1/2}\left(\tilde{\mathbf{L}}^{\dagger}-\mathbf{L}^{\dagger}\right)\mathbf{L}^{1/2}\right)^{2}}$$

where because $\mathbf{L}^{1/2}\left(\tilde{\mathbf{L}}^{\dagger}-\mathbf{L}^{\dagger}\right)\mathbf{L}^{1/2}$ is a symmetric matrix, we have by the definition of operator norm:

$$\leq\left\|\mathbf{L}^{1/2}\left(\tilde{\mathbf{L}}^{\dagger}-\mathbf{L}^{\dagger}\right)\mathbf{L}^{1/2}\right\|_{2}^{2}\left\|\mathbf{L}^{\dagger/2}\mathbf{b}\right\|_{2}=\left\|\mathbf{L}^{1/2}\left(\tilde{\mathbf{L}}^{\dagger}-\mathbf{L}^{\dagger}\right)\mathbf{L}^{1/2}\right\|_{2}^{2}\|\mathbf{b}\|_{\mathbf{L}^{\dagger}}. \tag{3.15}$$

Composing both sides of $\tilde{\mathbf{L}}\approx_{\epsilon}\mathbf{L}$ by $\mathbf{L}^{1/2}$ gives $\mathbf{L}^{1/2}\tilde{\mathbf{L}}\mathbf{L}^{1/2}\approx_{\epsilon}\mathbf{L}^{1/2}\mathbf{L}\mathbf{L}^{1/2}$, or upon rearranging:

$$-\epsilon\mathbf{I}\preceq\mathbf{L}^{1/2}(\tilde{\mathbf{L}}^{\dagger}-\mathbf{L}^{\dagger})\mathbf{L}^{1/2}\preceq\epsilon\mathbf{I},$$

or $\left\|\mathbf{L}^{1/2}(\tilde{\mathbf{L}}^{\dagger}-\mathbf{L}^{\dagger})\mathbf{L}^{1/2}\right\|_{2}^{2}\leq\epsilon$. Substituting this bound into Equation 3.15 above then gives the result.

Summing up these three cases as in Equation 3.14 then gives the overall result

$$\left\|\tilde{\mathbf{x}}-\mathbf{L}^{\dagger}\mathbf{b}\right\|_{\mathbf{L}}\leq 10\epsilon k.$$

□

3.5.1 Dynamic Projection

We next discuss the main ideas behind the dynamic algorithm that maintains an approximate projection in Lemma 3.5.2 and then formally describe the implementation of this data-structure together with its running time guarantees. To this end, suppose we are given an approximate projection $\tilde{\mathbf{b}}$ of $\mathbf{P}(S)\mathbf{b}$ satisfying the following inequality

$$\left\|\tilde{\mathbf{b}}-\mathbf{P}(S)\mathbf{b}\right\|_{\mathbf{L}^{\dagger}}\leq\epsilon\|\mathbf{b}\|_{\mathbf{L}^{\dagger}} \tag{3.16}$$

The crucial idea is to exploit the fact that the right hand side of the above inequality $\|\mathbf{b}\|_{\mathbf{L}^{\dagger}}$ corresponds to the square root of the energy needed by the electrical flow to route the demand vector $\mathbf{b}$ (see Lemma 2.1 in [194]). Since we assume that our dynamic graph G has bounded-degree, this energy is lower-bounded by

$$\|\mathbf{b}\|_{L^{\dagger}}\geq\sqrt{\sum_{u\in V}\left(\frac{|\mathbf{b}(u)|}{\mathbf{d}(u)}\right)}=\Omega\left(\sqrt{\sum_{u\in V}|\mathbf{b}(u)|}\right).$$

Let S' be the set of βm vertices such that their corresponding coordinates in $\mathbf{b}$ have the largest values. Without loss of generality, scale all the entries in the vector $\mathbf{b}$ such that

$$|\mathbf{b}(u)|\geq 1,\quad\forall u\in S'\quad\text{and}\quad|\mathbf{b}(u)|\leq 1,\quad\forall u\in V\setminus S' \tag{3.17}$$

By definition of S', after up to $(\beta m)/2$ operations in our data-structure, we know that at least half of the vertices in S' will keep their corresponding $\mathbf{b}$ values. Thus the allowable error from right hand side of Equation (3.16), $\|\mathbf{b}\|_{\mathbf{L}^\dagger}$, is lower bounded by $\Omega(\sqrt{\beta m})$. Our goal is to control the error between the maintained approximate projection $\tilde{\mathbf{b}}$ and the true projection $\mathbf{P}(S)\mathbf{b}$. Our algorithm has two main components. First, it shows how to use a Laplacian solver that computes an approximate projection $\tilde{\mathbf{b}}$ of $\mathbf{P}(S)\mathbf{b}$ satisfying Equation (3.16) in nearly-linear time. Second, it gives a way to control the error of the projection $\mathbf{P}(S)\mathbf{b}$ under terminal additions to S with respect to the $\|\cdot\|_{\mathbf{L}^\dagger}$ norm.

We now state the initialization lemma, whose is proof is deferred to Subsection 3.5.2.

Lemma 3.5.4. *Given an unweighted graph $G = (V, E)$ with n vertices and m edges, a demand vector $\mathbf{b} \in \mathbb{R}^n$, set of vertices $S \subseteq V$ and an error parameter $\epsilon > 0$, there is an $\tilde{O}(m)$ time algorithm that computes a vector $\tilde{\mathbf{b}}$ such that*

$$\left\|\tilde{\mathbf{b}} - \mathbf{P}(S)\mathbf{b}\right\|_{\mathbf{L}^\dagger} \leq \epsilon \|\mathbf{b}\|_{\mathbf{L}^\dagger}.$$

To elaborate on the second component of the algorithm, consider the error induced on $\mathbf{P}(S)\mathbf{b}$ when we add a vertex u to some terminal set S

$$\left\|\mathbf{P}(S)\mathbf{b} - \mathbf{P}(\tilde{S})\mathbf{b}\right\|_{\mathbf{L}^\dagger}, \quad \text{where } \tilde{S} = S \cup \{u\}.$$

In other words, the above expression gives the error when we simply keep the same vector $\mathbf{b}$ under a terminal addition to the set S. We will show that over a certain number of such additions we can bound the compounded error by $O(\sqrt{\beta m})$. Since the latter is a lower bound on $\|\mathbf{b}\|_{\mathbf{L}^\dagger}$, it follows that the maintained projection still provide good approximation guarantee. The following lemma, whose proof is deferred to Subsection 3.5.3, bounds the error after one terminal addition.

Lemma 3.5.5. *Consider an unweighted undirected bounded-degree graph $G = (V, E)$, a demand vector $\mathbf{b} \in \mathbb{R}^n$ and a parameter $\beta \in (0, 1)$. Let $S \subseteq V$ with $|S| = O(\beta m)$ and assume that $|\mathbf{b}(u)| \geq 1$ for all $u \in S$, and $|\mathbf{b}(u)|$ for all $u \in V \setminus S$. For each edge in G, include its endpoints to S independently, with probability at least β. Then, for any vertex $u \in V \setminus S$, with high probability*

$$\left\|\mathbf{P}(S)\mathbf{b} - \mathbf{P}(\tilde{S})\mathbf{b}\right\|_{\mathbf{L}^\dagger} = \tilde{O}(\beta^{-5/2}), \quad \textit{where } \tilde{S} = S \cup \{u\}.$$

We now have all the necessary tools to give a dynamic data-structure that maintains an approximate projection, i.e., prove Lemma 3.5.2.

Proof of Lemma 3.5.2. Given the input demand vector $\mathbf{b}$, let S' be the set of βm vertices such that their corresponding coordinates in $\mathbf{b}$ have the largest values. Without loss of generality, scale $\mathbf{b}$ according to Equation (3.17). For each edge in G include its endpoints to S' independently, with probability at least β.

We next describe the implementation of the operations. For implementing procedure INITIALIZE(G, S', β), we invoke Lemma 3.5.4 with $\epsilon/2$. Let $\tilde{\mathbf{b}}$ be the output approximate projection satisfying Equation (3.16) with error parameter $\epsilon/2$ and set $S = S'$. As we will shortly see, operations INSERT and DELETE will be reduced to adding terminals to the set S. Thus we first discuss the implementation of the operation ADDTERMINAL. To this end, let u be a non-terminal vertex that we want to append to S. We set $S = S \cup \{u\}$ and simply add an entry $\tilde{\mathbf{b}}(u) = 0$ to $\tilde{\mathbf{b}}$ while keeping the rest of the entries unaffected. To insert or delete an edge from the current graph, we simply run ADDTERMINAL procedure for the edge endpoints.

Consider the operation CHANGE$(u, \mathbf{b}(u)', v, \mathbf{b}(v)')$. We first invoke ADDTERMINAL on both u and v and then add $\mathbf{b}(u)' - \mathbf{b}(u)$ to $\tilde{\mathbf{b}}(u)$ and $\mathbf{b}(v)' - \mathbf{b}(v)$ to $\tilde{\mathbf{b}}(v)$. Finally, to implement QUERY we simply return the approximate projection $\tilde{\mathbf{b}}$.

We next analyze the correctness of our data-structure which solely depends on the correctness of ADDTERMINAL and CHANGE operations. We will show that after k many such operations, our maintained approximate projection $\tilde{\mathbf{b}}$ satisfies

$$\left\|\tilde{\mathbf{b}} - \mathbf{P}(S)\mathbf{b}\right\|_{\mathbf{L}^\dagger} \leq \tilde{O}(k\beta^{-5/2}) + (\epsilon/2)\left\|\mathbf{b}\right\|_{\mathbf{L}^\dagger}, \tag{3.18}$$

where S denotes the set of terminals after k operations. Note that when $(k = 0)$, the above inequality holds by Lemma 3.5.4 that implements the initialization. Let us analyze the error when a single terminal is added to S, i.e., $(k = 1)$. Then Lemma 3.5.5 implies that

$$\left\|\mathbf{P}(S)\mathbf{b} - \mathbf{P}(\tilde{S})\mathbf{b}\right\|_{\mathbf{L}^\dagger} = \tilde{O}(\beta^{-5/2}), \quad \text{where } \tilde{S} = S \cup \{u\}.$$

Combining these two guarantees and applying triangle inequality, we get that the error after one terminal addition is

$$\begin{aligned}\left\|\tilde{\mathbf{b}} - \mathbf{P}(\tilde{S})\mathbf{b}\right\|_{\mathbf{L}^\dagger} &= \left\|\tilde{\mathbf{b}} - \mathbf{P}(\tilde{S})\mathbf{b} + \mathbf{P}(S)\mathbf{b} - \mathbf{P}(S)\mathbf{b}\right\|_{\mathbf{L}^\dagger} \\ &\leq \left\|\tilde{\mathbf{b}} - \mathbf{P}(S)\mathbf{b}\right\|_{\mathbf{L}^\dagger} + \left\|\mathbf{P}(S)\mathbf{b} - \mathbf{P}(\tilde{S})\mathbf{b}\right\|_{\mathbf{L}^\dagger} \\ &\leq \tilde{O}(\beta^{-5/2}) + (\epsilon/2)\left\|\mathbf{b}\right\|_{\mathbf{L}^\dagger}.\end{aligned}$$

Next, to analyze the changes in the values of $\mathbf{b}$, let the updated $\mathbf{b}'$ be the updated vector $\mathbf{b}$. Let $\tilde{\mathbf{b}}'$ be the updated $\tilde{\mathbf{b}}$ and let $\mathbf{P}(S)\mathbf{b}'$ be the updated $\mathbf{P}(S)\mathbf{b}$. Using the fact that u and v are added to S, we get that

$$(\tilde{\mathbf{b}} - \tilde{\mathbf{b}}') = (\mathbf{b} - \mathbf{b}') = (\mathbf{P}(S)\mathbf{b} - \mathbf{P}(S)\mathbf{b}'),$$

which in turn implies that

$$(\tilde{\mathbf{b}} - \mathbf{P}(S)\mathbf{b}) = (\tilde{\mathbf{b}}' - \mathbf{P}(S)\mathbf{b}'),$$

and thus the error vector does not change.

We showed that after each operation, either the correct vector moves by at most $\tilde{O}(\beta^{-5/2})$ with respect to its $\|\cdot\|_{\mathbf{L}^\dagger}$ norm, or $(\tilde{\mathbf{b}} - \mathbf{P}(S)\mathbf{b})$ does not change. Thus repeating the above argument k times yields Equation (3.18). Setting $k = c_{\text{EN}} \cdot m^{1/2}\beta^3\epsilon(\text{poly}\log n)^{-1})$ such that $\|\mathbf{b}\|_{\mathbf{L}^\dagger} = c_{\text{EN}} \cdot \sqrt{\beta m}$, we get that

$$\left\|\tilde{\mathbf{b}} - \mathbf{P}(S)\mathbf{b}\right\|_{\mathbf{L}^\dagger} \leq (\epsilon/2)c_{\text{EN}}\sqrt{\beta m} + (\epsilon/2)\left\|\mathbf{b}\right\|_{\mathbf{L}^\dagger} \leq \epsilon\left\|\mathbf{b}\right\|_{\mathbf{L}^\dagger}.$$

For the running time, Lemma 3.5.4 implies that the initialization cost is bounded by $\tilde{O}(m)$. Since the size of the maintained vector $\tilde{\mathbf{b}}$ is bounded by $|S|$, it follows that the query cost is $O(\beta m)$. All the remaining operations can be implemented in $O(1)$ time. □

3.5.2 Initialization of Approximate Projection Vector

In this subsection we show how to compute an initial approximate projection vector of $\mathbf{P}(S)\mathbf{b}$, i.e., we prove Lemma 3.5.4.

Proof of Lemma 3.5.4. Define $F = V \setminus S$ and let G' be an n'-vertex graph obtained from G by contracting all vertices in S within G into a single vertex s and keeping parallel edges. Let $\mathbf{L}'$ denote the corresponding Laplacian matrix of G' and consider the induced vertex mapping $\gamma : V \to V(G')$ with $\gamma(u) = u$ for $u \in F$ and $\gamma(u) = s$ for $u \in S$. Let $\mathbf{b}' \in \mathbb{R}^{n'}$ be the corresponding demand vector in G' such that for $u \in V$, $\mathbf{b}'(\gamma(u)) = \mathbf{b}(u)$ if $\gamma(u) = u$ and $\mathbf{b}'(\gamma(u)) = \sum_{v\in S}\mathbf{b}(v)$ otherwise. For the given error parameter $\epsilon > 0$, we can invoke a black-box Laplacian solver to compute an approximate solution vector $\tilde{\mathbf{v}}'$ to $\mathbf{v}' = \mathbf{L}'^{\dagger}\mathbf{b}'$ such that

$$\left\|\tilde{\mathbf{v}}' - \mathbf{v}'\right\|_{\mathbf{L}'} \leq \epsilon\left\|\mathbf{v}'\right\|_{\mathbf{L}'}. \tag{3.19}$$

Now, to lift back the vector $\tilde{\mathbf{v}}'$ to G we define new vectors $\tilde{\mathbf{v}}$ and $\mathbf{v}$ such that for all $u \in V$

$$\tilde{\mathbf{v}}(u) := \tilde{\mathbf{v}}'(\gamma(u)) \text{ and } \mathbf{v}(u) := \mathbf{v}'(\gamma(u)).$$

Observe that for any edge $e = (u, v)$ in G, we have that

$$(\tilde{\mathbf{v}}(u) - \tilde{\mathbf{v}}(v)) = (\tilde{\mathbf{v}}'(u) - \tilde{\mathbf{v}}(v)) \text{ and } (\mathbf{v}(u) - \mathbf{v}(v)) = (\mathbf{v}'(u) - \mathbf{v}'(v)).$$

The above relations imply that the approximation guarantee from Equation (3.19) can be written as follows

$$\left\|\tilde{\mathbf{v}} - \mathbf{v}\right\|_{\mathbf{L}} \leq \epsilon\left\|\mathbf{v}\right\|_{\mathbf{L}}. \tag{3.20}$$

It is well known that if we interpret G as a resistor network, $\mathbf{v}$ represents the voltage vector on the vertices induced by the electrical flow that routes a certain demand in the network (see e.g., [86]). Thus, by linearity of electrical flows and our construction, we can view $\mathbf{v}$ as being the sum of the voltage vectors corresponding to the electrical flows that route $\mathbf{b}(u)$ amount of flow to S, where the sum is over all

$u \in F$. By Lemma 3.2.8, for each $u \in F$, the demand corresponding to the electrical flow that send $\mathbf{b}(u)$ units of flow to S is given by

$$\mathbf{b}(u)(\mathbf{1}_u - \mathbf{P}(S)\mathbf{1}_u).$$

Summing over all $u \in F$ we get the demand vector corresponding to $\mathbf{v}$

$$\sum_{u \in F} \mathbf{b}(u)(\mathbf{1}_u - \mathbf{P}(S)\mathbf{1}_u) = (\mathbf{b}|_F - \mathbf{P}(S)\mathbf{b}|_F) = (\mathbf{b}|_F - \mathbf{P}(S)(\mathbf{b} - \mathbf{b}|_S))$$
$$= (\mathbf{b}|_F - \mathbf{P}(S)\mathbf{b} - \mathbf{b}|_S)\,,$$

where $\mathbf{b}|_U$ is the restriction of $\mathbf{b}$ on the subset U with $\mathbf{b}|_U(u) = \mathbf{b}(u)$ if $u \in U$, and $\mathbf{b}|_U(u) = 0$ otherwise. Since we determined the demand vector corresponding to $\mathbf{v}$, we get that

$$\mathbf{L}\mathbf{v} = (\mathbf{b}|_F - \mathbf{P}(S)\mathbf{b} - \mathbf{b}|_S)\,. \tag{3.21}$$

Define the approximate project vector $\tilde{\mathbf{b}}$ that our algorithm outputs using the following relation

$$\tilde{\mathbf{b}} := (\mathbf{b}|_F - \mathbf{L}\tilde{\mathbf{v}} - \mathbf{b}|_S)\,, \tag{3.22}$$

where $\tilde{\mathbf{v}}$ is the extended voltage vector we defined above. To complete the proof of the lemma, it remains to bound the difference between $\tilde{\mathbf{b}}$ and $\mathbf{P}(S)\mathbf{b}$ with respect to the $\mathbf{L}^\dagger$ norm. To this end, using Equations (3.21) and (3.22) we have

$$\left\|\tilde{\mathbf{b}} - \mathbf{P}(S)\mathbf{b}\right\|_{\mathbf{L}^\dagger} = \|\mathbf{b}|_F - \mathbf{L}\tilde{\mathbf{v}} - \mathbf{b}|_S - (\mathbf{b}|_F - \mathbf{L}\mathbf{v} - \mathbf{b}|_S)\|_{\mathbf{L}^\dagger}$$
$$= \|\mathbf{L}\tilde{\mathbf{v}} - \mathbf{L}\mathbf{v}\|_{\mathbf{L}^\dagger} = \|\tilde{\mathbf{v}} - \mathbf{v}\|_{\mathbf{L}}\,.$$

Using the approximate guarantee in Equation (3.20) we have that

$$\|\tilde{\mathbf{v}} - \mathbf{v}\|_{\mathbf{L}} \leq \epsilon \|\mathbf{v}\|_{\mathbf{L}} = \epsilon \left\|\mathbf{v}'\right\|_{\mathbf{L}'} = \epsilon \left\|\mathbf{b}'\right\|_{\mathbf{L}'^\dagger} \leq \epsilon \|\mathbf{b}\|_{\mathbf{L}^\dagger}\,,$$

where the last inequality follows from the fact that the minimum energy needed to route $\mathbf{b}$ becomes smaller when contracting vertices. □

3.5.3 Stability of Projected Vectors

In this subsection we prove our core structural observation, namely that the the projection vectors remain stable under the addition of a new terminal vertex, as stated in Lemma 3.5.5.

We start by considering the projection vector $\mathbf{P}(S)\mathbf{1}_u$, where $u \in F = V \setminus S$. Recall that for $s \in S$, Lemma 3.2.7 gives that $[\mathbf{P}(S)\mathbf{1}_u](s)$ is the probability that the random walk that starts at u hits the set S at the vertex s. Equivalently, we can view the probability of this walk as routing a fraction of $\mathbf{1}_u$ from u to s. Now, consider the operation of adding a non-terminal $u \in F$ to S, i.e., $\tilde{S} = S \cup \{u\}$. We observe that the fraction of $\mathbf{1}_u$ that we routed to some vertex v in S might have used the vertex $u \in F$. This indicates that this this fraction should have stopped at u,

instead of going to other vertices in S, which in turn implies that the old projection vector $\mathbf{P}(S)\mathbf{1}_u$ is not valid anymore. We will later show that this change is tightly related to the load that random walks from other vertices in F put on the new terminal vertex u. In the following we focus on showing a provable bound on this load, which in turn will allow us to control the error for the maintained projection vector.

Concretely, for each vertex $u \in F$, we want to bound the load incurred by the random walks of the other vertices $v \in F \setminus u$ to the set S. For the purposes of our proof, it will be useful to introduce some random variable. For $v \in F$, let $Z_v(S)$ be the set of vertices visited in a random walk starting at v and ending at some vertex in S. For $t \geq 0$, let $X_v(t)$ be the set of vertices visited in a random walk starting at $v \in F$ after K steps. For a demand vector $\mathbf{b}$ and any two vertices $u, v \in F$, the contribution of v to the load of u, denoted by $Y_v(u)$, is defined as follows

$$Y_v(u) = \mathbf{b}(v) \cdot \mathbf{1}_{(u \in Z_v(S))}.$$

The *load* of a vertex $u \in F$, denoted by N_u, is obtained by summing the contributions over all vertices in F, i.e.,

$$N_u = \sum_{v \in F} Y_v(u).$$

The following lemma gives a bound on the expected load of every non-terminal vertex.

Lemma 3.5.6. *For a parameter $\beta \in (0, 1)$ and every vertex $u \in F$ we have that $\mathbb{E}[N_u] = \tilde{O}(\beta^{-2})$.*

For proving the above lemma it will be useful to rewrite the load quantity. To this end, recall that in the proof of Theorem 3.4.3 we have shown that any random walk that start at a vertex v of length $\ell = \tilde{O}(\beta^{-2})$ hits a vertex in the terminal set K with probability at least $1 - 1/n^c$, for some large constant c. Note that by construction of S in Lemma 3.5.5, the exact same argument applies to the set S. Thus, instead of terminating the random walks once they hit S, we can run all the walks from the vertices in F up to ℓ steps. The latter together with the assumption $\mathbf{b}(v) \leq 1$ for all $v \in F$ (provided by Lemma 3.5.5) give that

$$\begin{aligned}
\mathbb{E}[N_u] &= \sum_{v \in F} \mathbf{b}(v) \cdot \mathbb{P}_v[v \in Z_v(S)] \\
&\leq \sum_{v \in F} \left(\mathbb{P}_v[\text{walk } w \text{ from } v \text{ uses } u \text{ in its first } \ell \text{ steps}] + \mathbb{P}_v[|w| > \ell]\right) \\
&\leq \sum_{v \in F} \left(\sum_{0 \leq t \leq \ell} \mathbb{P}_v[u \in X_v(t)] + 1/n^c \right) \\
&\leq \sum_{0 \leq t \leq \ell} \left(\sum_{v \in V} \deg(v) \cdot \mathbb{P}_v[u \in X_v(t)] \right) + o(1). \qquad (3.23)
\end{aligned}$$

It turns out that that the term contained in the brackets of Equation (3.23) equals $\deg(u)$. Formally, we have the following lemma.

Lemma 3.5.7. *Let G be an undirected unweighted graph. For any vertex $u \in V$ and any length $t \geq 0$, we have*

$$\sum_{v \in V} \deg(v) \cdot \mathbb{P}\left[u \in X_v(t)\right] = \deg(u).$$

To prove this, we use the reversibility of random walks, along with the fact that the total probability over all edges of a walk starting at e is 1 at any time. Below we verify this fact in a more principled manner.

Proof of Lemma 3.5.7. The proof is by induction on the length of the walks K. When $t = 0$, we have

$$\mathbb{P}\left[u \in X_v(0)\right] = \begin{cases} 1 & \text{if } u = v, \\ 0 & \text{otherwise,} \end{cases}$$

which gives a total of $\deg(u)$.

For the inductive case, assume the result is true for $t-1$. The probability of a walk reaching u after K steps can then be written in terms of its location at time $t-1$, the neighbor x of u, as well as the probability of reaching there:

$$\mathbb{P}\left[u \in X_v(t)\right] = \sum_{x:(u,x)\in E} \frac{1}{\deg(x)} \mathbb{P}\left[x \in X_v(t-1)\right].$$

Substituting this into the summation to get

$$\sum_{v \in V} \deg(v) \cdot \mathbb{P}\left[u \in X_v(t)\right] = \sum_{v \in V} \deg(v) \sum_{x:(u,x)\in E} \frac{1}{\deg(x)} \mathbb{P}\left[x \in X_v(t-1)\right],$$

which upon rearranging of the two summations gives:

$$\sum_{x:(u,x)\in E} \frac{1}{\deg(x)} \left(\sum_{v \in V} \deg(v) \cdot \mathbb{P}\left[x \in X_v(t-1)\right] \right).$$

By the inductive hypothesis, the term contained in the bracket is precisely $\deg(x)$, which cancels with the division, and leaves us with $\deg(u)$. Thus the inductive hypothesis holds for K as well. □

Plugging Lemma 3.5.7 in Equation (3.23), along with the fact that by assumption G has bounded degree we get that

$$\mathbb{E}\left[N_u\right] \leq \deg(u) \cdot \ell = \tilde{O}(\beta^{-2}),$$

thus proving Lemma 3.5.6.

We now have all the tools to prove Lemma 3.5.5.

Proof of Lemma 3.5.5. Recall that $\tilde{S} = S \cup \{u\}$, where u is vertex in $F = V \setminus S$. We want to obtain a bound on the difference $(\mathbf{P}(S)\mathbf{b} - \mathbf{P}(\tilde{S})\mathbf{b})$ with respect to the $\mathbf{L}^\dagger$ norm. We distinguish the following types of entries of the difference vector: (1) newly added terminal u, (2) the old terminals S and (3) the remaining non-terminal vertices $F \setminus \{u\}$. Note that $\mathbf{P}(S)\mathbf{b}$ and $\mathbf{P}(\tilde{S})\mathbf{b}$ are not n-dimensional vectors, so we assume that all missing entries are appended with zeros. This also allows us to compute the $\mathbf{L}^\dagger$ norm.

In what follows, we will repeatedly make of the following relation by Lemma 3.2.7 for vertices $u \in F$ and $v \in S$

$$\mathbf{P}(S)\mathbf{1}_u(v) = \sum_{\substack{u_0=u,\ldots,u_{k-1}\in F,\\ u_k=v}} \frac{\prod_{i=0}^{k-1} \mathbf{w}(u_i, u_{i+1})}{\prod_{i=1}^{k-1} \mathbf{d}(u_i)}.$$

For the type (1) entry, i.e., newly added terminal u, using the definition of the load N_u, we get:

$$\begin{aligned}[\mathbf{P}(S)\mathbf{b} - \mathbf{P}(\tilde{S})\mathbf{b}](u) &= -\sum_{\substack{u_0=u,\ldots,u_{k-1}\in F\setminus\{u\},\\ u_k=u}} \mathbf{b}(u_0) \cdot \frac{\prod_{i=0}^{k-1} \mathbf{w}(u_i, u_{i+1})}{\prod_{i=1}^{k-1} \mathbf{d}(u_i)} \\ &= -\sum_{u_0\in F} \mathbb{E}\left[Y_{u_0}(u)\right] = -\mathbb{E}\left[N_u\right]. \end{aligned} \tag{3.24}$$

Note that for type (3) entries, i.e., the remaining non-terminals $v \in F \setminus \{u\}$, we have that

$$[\mathbf{P}(S)\mathbf{b} - \mathbf{P}(\tilde{S})\mathbf{b}](v) = 0. \tag{3.25}$$

Finally, for type (2) entries, i.e., old terminals $v \in S$, similarly to the type (1) entries we get

$$\begin{aligned}[\mathbf{P}(S)\mathbf{b} - \mathbf{P}(\tilde{S})\mathbf{b}](v) &= \sum_{\substack{u_0=u,\ldots,u_{k-1}\in F,\\ u_k=v}} \mathbf{b}(u_0) \cdot \frac{\prod_{i=0}^{k-1} \mathbf{w}(u_i, u_{i+1})}{\prod_{i=1}^{k-1} \mathbf{d}(u_i)} \\ &\quad - \sum_{\substack{u_0=u,\ldots,u_{k-1}\in F\setminus\{u\},\\ u_k=v}} \mathbf{b}(u_0) \cdot \frac{\prod_{i=0}^{k-1} \mathbf{w}(u_i, u_{i+1})}{\prod_{i=1}^{k-1} \mathbf{d}(u_i)} \\ &= \sum_{\substack{u_0=u,\ldots,u_k=u,\\ u_{k+1},\ldots,u_{r-1}\in F, u_r=v}} \mathbf{b}(u_0) \cdot \frac{\prod_{i=0}^{r-1} \mathbf{w}(u_i, u_{i+1})}{\prod_{i=1}^{r-1} \mathbf{d}(u_i)} \\ &= \sum_{\substack{u_0=u,\ldots,u_{k-1}\in F\setminus\{u\},\\ u_k=v}} \mathbf{b}(u_0) \cdot \frac{\prod_{i=0}^{k-1} \mathbf{w}(u_i, u_{i+1})}{\prod_{i=1}^{k-1} \mathbf{d}(u_i)} \\ &\quad \sum_{\substack{u_0=u,\ldots,u_{k-1}\in F,\\ u_k=v}} \frac{\prod_{i=0}^{k-1} \mathbf{w}(u_i, u_{i+1})}{\prod_{i=1}^{k-1} \mathbf{d}(u_i)} \end{aligned}$$

$$= \mathbb{E}[N_u] \cdot [\mathbf{P}(S)\mathbf{1}_u](v). \tag{3.26}$$

Bringing together Equations (3.24), (3.25) and (3.26) we get that

$$[\mathbf{P}(S)\mathbf{b} - \mathbf{P}(\tilde{S})\mathbf{b}] = -(\mathbb{E}[N_u](\mathbf{1}_u - \mathbf{P}(S)\mathbf{1}_u)).$$

The right-hand side of the equation can be interpreted as routing $\mathbb{E}[N_u]$ unit of flows from u to S. Thus, to measure the error, we simply need to upper-bound the square root of the energy need to route $\mathbb{E}[N_u]$ amount of flow from u to S (Lemma 3.2.8),i.e.,

$$\|\mathbb{E}[N_u](\mathbf{1}_u - \mathbf{P}(S)\mathbf{1}_u)\|_{\mathbf{L}^\dagger}.$$

By the simplifying assumption that G is connected and the fact that each endpoint of an edge in E is added to S independently, with probability at least β, it is easy to show that with high probability, there exists a path $p(v, S)$ from u to S that uses at most $O(\beta^{-1}\log n)$ edges. Hence, if we route $\mathbb{E}[N_u]$ units of flow from u to S along the path $p(v, S)$, the energy of such a flow is upper-bounded by

$$(\mathbb{E}[N_u])^2 \cdot \tilde{O}(\beta^{-1}) = \tilde{O}((\mathbb{E}[N_u])^2\beta^{-1}).$$

Using the latter we get that

$$\begin{aligned}\left\|\mathbf{P}(S)\mathbf{b} - \mathbf{P}(\tilde{S})\mathbf{b}\right\|_{\mathbf{L}^\dagger} &= \|\mathbb{E}[N_u](\mathbf{1}_u - \mathbf{P}(S)\mathbf{1}_u)\|_{\mathbf{L}^\dagger} \\ &\leq \tilde{O}\left(\sqrt{(\mathbb{E}[N_u])^2\beta^{-1}}\right) \\ &= \tilde{O}(\mathbb{E}[N_u]\,\beta^{-1/2}) \\ &= \tilde{O}(\beta^{-5/2}),\end{aligned}$$

where the last inequality uses the fact that $\mathbb{E}[N_u] = \tilde{O}(\beta^{-2})$ by Lemma 3.5.6. This completes the proof the lemma. □

3.6 Sampling Weights of a Random Walk

In this section, we show that given a random walk w of length ℓ in a weighted G with polynomially bounded weights, we can efficiently sample an approximation to $s(w) = \sum_{i=1}^{\ell}(1/\mathbf{w}(w_{i-1}, w_i))$. Concretely, we prove the following lemma from Section 3.4.3.

Lemma 3.6.1 (Restatement of Lemma 3.4.13)**.** *Let $G = (V, E, \mathbf{w})$ be a undirected, weighted graph with $\mathbf{w}(e) = [1, n^c]$ for each $e \in E$, where c is a positive constant. For any finite random walk w of length ℓ with $\ell \leq n^d$, where d is a positive constant, let $s(w)$ be the sum of the inverse of its edge weights, i.e.,*

$$s(w) = \sum_{i=1}^{\ell} \frac{1}{\mathbf{w}(w_{i-1}, w_i)}.$$

Moreover, for any $u, v \in V$, let

$$f^{u,v}_{s(w),\ell}$$

be the probability mass function of $s(w)$ conditioning on (1) w being a random walk that starts at u and ends at v, and (2) length of the walk $\ell(w)$ is ℓ in G. Then, for any pair $u, v \in V$, there exists an algorithm that that samples from $f^{u,v}_{s(w),\ell}$ and outputs a sampled $s(w)$ up to a $(1+\epsilon)$ multiplicative error in $\tilde{O}(n^3\epsilon^{-2})$ time.

To prove the above lemma, we employ a doubling technique. Specifically, for any pair of vertices $u, v \in V$, and a random walk w of length ℓ that starts at u and ends at v, it is easy to see that

$$f^{u,v}_{s(w),\ell} = \sum_{y \in V} \left(f^{u,y}_{s(w),\ell/2} * f^{y,v}_{s(w),\ell/2} \right),$$

where $*$ denotes the convolution between two probability mass functions. However, one challenge here is that we cannot afford dealing with exact representations of probability mass functions as this would be computationally expensive. Instead, we introduce an *approximate* representation of such functions, and then give an algorithm that allows computing the convolution between such approximate representations. Before proceeding further, note that we can scale down the edge weights so that $\mathbf{w}(e) \leq 1$, and thus $1/\mathbf{w}(e) \geq 1$ for every $e \in E$. In addition, we remark that $\mathbf{w}$ does not need to be integral.

Let us introduce a compact way to represent any given probability mass function approximately f. The main idea is to 'move' each number in the support of f by $(1+\epsilon)$, which in turn results in a $(1+\epsilon)$ approximation of the sampled value for f. Formally, let f be a probability mass function such that $f(x) = 0$, for each $x \notin \{0, \ldots, n^c\}$, where c is a positive constant. For $j \geq 1$, let I^j_k be the interval $[(1+\epsilon)^k, (1+\epsilon)^{k+j})$ for $k \in \{0, \ldots, L\}$ where $L = O\left((c+d)\epsilon^{-1}\log n\right)$. Note that the upper bound L is chosen in such a way that $\cup_k I^1_k$ covers the range of $f^{u,\ell}_{s(w),\ell}$ for every possible triplet (u, v, ℓ). For $j \geq 1$ and $\epsilon > 0$, we say that g is an (ϵ, j)-approximation of a probability mass function f iff there exists a matrix $\mathbf{H}$ satisfying the following properties:

(a) $\sum_{k=0}^{L} \mathbf{H}_{x,k} = f(x),\ \forall x \in \{0, \ldots, n^c\}$,

(b) $\sum_{x=0}^{n^c} \mathbf{H}_{x,k} = g(k),\ \forall k \in \{0, \ldots, L\}$,

(c) $\mathbf{H}_{x,k} = 0,\ \forall x \notin I^j_k$.

Note that an (ϵ, j)-approximation of f is also an $(\epsilon, (j+1))$-approximation of f. Moreover, observe that the intervals $\{I^1_k\}_{k \in \{0\} \cup L}$ are disjoint for different k but I^j_k overlaps with $I^j_{k'}$ whenever $j \geq 2$ and $|k - k'| < j$.

Algorithm 3.6: CONVOLUTE$(g^{(1)}, g^{(2)}, \epsilon, j)$

Input : Two (ϵ, j)-approximations $g^{(1)}$ and $g^{(2)}$ of two probability mass functions $f^{(1)}$ and $f^{(2)}$

Output : An $(\epsilon, (j+1))$-approximation $g := g^{(1)} * g^{(2)}$ of $f := f^{(1)} * f^{(2)}$

```
Set g ← 0
for (k_1, k_2) ∈ {0, ..., L}^2 do
    Find k_3 such that (1+ε)^{k_1} + (1+ε)^{k_2} ∈ I^1_{k_3}
    Set g(k_3) ← g(k_3) + g^{(1)}(k_1) · g^{(2)}(k_2)
return g
```

Next we show how to compute the convolution of two probability mass functions under their approximate representations. Let and $g^{(1)}$ and $g^{(2)}$ be (ϵ, j)-approximations of probability mass functions $f^{(1)}$ and $f^{(2)}$, respectively. Now consider two intervals $I^j_{k_1}$ and $I^j_{k_2}$. Without loss of generality, assume that $k_1 \leq k_2$. If $x \in I^j_{k_1}$ and $y \in I^j_{k_2}$, then

$$x + y \in I' := [\mathsf{le}, \mathsf{ri}), \text{ where}$$

$$\mathsf{le} := \left((1+\epsilon)^{k_1} + (1+\epsilon)^{k_2}\right), \quad \mathsf{ri} := \left((1+\epsilon)^{k_1+j} + (1+\epsilon)^{k_2+j}\right).$$

Furthermore, let $I^1_{k_3}$ be an interval such that $\mathsf{le} \in I^1_{k_3}$. The latter implies that $(1+\epsilon)^{k_3} \leq \mathsf{le} < (1+\epsilon)^{k_3+1}$. Since $\mathsf{ri} = \mathsf{le} \cdot (1+\epsilon)^j$, it follows that $\mathsf{ri} < (1+\epsilon)^{k_3+j+1}$. Bringing together the above bounds we get that $(1+\epsilon)^{k_3} \leq \mathsf{le} < (1+\epsilon)^{k_3+j+1}$, i.e., $I' \subseteq I^{j+1}_{k_3}$. Since k_3 depends on k_1, k_2, and j we sometimes write $k_3(k_1, k_2, j)$ instead of k_3.

Since the above approach gives us a way to combine two different intervals, it is now straightforward to compute the convolution between two probability mass functions. This task is performed in the standard way and we review its implementation details in Algorithm 3.6 for the sake of completeness.

Lemma 3.6.2. *Let $j \geq 1$ and $\epsilon > 0$ by two parameters. Given any two (ϵ, j)-approximations $g^{(1)}$ and $g^{(2)}$ of probability mass functions $f^{(1)}$ and $f^{(2)}$, CONVOLUTE$(g^{(1)}, g^{(2)}, \epsilon, j)$ (Algorithm 3.6) computes in $\tilde{O}(\epsilon^2)$ time an $(\epsilon, (j+1))$-approximation $g := g^{(1)} * g^{(2)}$ of the convolution $f := f^{(1)} * f^{(2)}$.*

Proof. We first show the correctness. Since $g^{(1)}$ and $g^{(2)}$ are (ϵ, j)-approximations to $f^{(1)}$ and $f^{(2)}$ by assumption of the lemma, we know that there exists matrices $\mathbf{H}^{(1)}$ and $\mathbf{H}^{(2)}$ satisfying properties (a), (b) and (c). To show that the output g is correct we need to construct a matrix $\mathbf{H}$ that satisfies each of these properties. By construction of the algorithm, the new matrix $\mathbf{H}$ is defined as follows:

$$\mathbf{H}_{z,k_3} := \sum_{\substack{x \in I^j_{k_1}, x \in I^j_{k_2} \\ x+y=z, k_3 = k_3(k_1,k_2,j)}} \mathbf{H}^{(1)}_{x,k_1} \cdot \mathbf{H}^{(2)}_{y,k_2}, \quad z \in \{0, \ldots, n^c\},\ k_3 \in \{0, \ldots, L\}.$$

We start by showing property (a) for $\mathbf{H}$. Concretely, for any $z \in \{0, \ldots, n^c\}$ we get that

$$\begin{aligned}
\sum_{k_3=0}^{L} \mathbf{H}_{z,k_3} &= \sum_{k_3=0}^{L} \sum_{\substack{x \in I^j_{k_1}, x \in I^j_{k_2} \\ x+y=z, k_3=k_3(k_1,k_2,j)}} \mathbf{H}^{(1)}_{x,k_1} \cdot \mathbf{H}^{(2)}_{y,k_2} \\
&= \sum_{x \in I^j_{k_1}, y \in I^j_{k_2}, x+y=z} \mathbf{H}^{(1)}_{x,k_1} \cdot \mathbf{H}^{(2)}_{y,k_2} \\
&= \sum_{x+y=z} \left(\sum_{x \in I^j_{k_1}} \mathbf{H}^{(1)}_{x,k_1} \right) \left(\sum_{y \in I^j_{k_2}} \mathbf{H}^{(2)}_{y,k_2} \right) \\
&= \sum_{x+y=z} f^{(1)}(x) \cdot f^{(2)}(y) \\
&= \left(f^{(1)} * f^{(2)} \right)(z) = f(z).
\end{aligned}$$

Next, $\mathbf{H}$ satisfies property (b) since for any $k_3 \in \{0, \ldots, L\}$ we get that

$$\begin{aligned}
\sum_{z=0}^{n^c} \mathbf{H}_{z,k_3} &= \sum_{z=0}^{n^c} \sum_{\substack{x \in I^j_{k_1}, x \in I^j_{k_2} \\ x+y=z, k_3=k_3(k_1,k_2,j)}} \mathbf{H}^{(1)}_{x,k_1} \cdot \mathbf{H}^{(2)}_{y,k_2} \\
&= \sum_{x \in I^j_{k_1}, y \in d^j_{k_2}, k_3=k_3(k_1,k_2,j)} \mathbf{H}^{(1)}_{x,k_1} \cdot \mathbf{H}^{(2)}_{y,k_2} \\
&= \sum_{k_3=k_3(k_1,k_2,j)} \left(\sum_{x \in I^j_{k_1}} \mathbf{H}^{(1)}_{x,k_1} \right) \left(\sum_{y \in I^j_{k_2}} \mathbf{H}^{(2)}_{y,k_2} \right) \\
&= \sum_{k_3=k_3(k_1,k_2,j)} g^{(1)}_{k_1} \cdot g^{(2)}_{k_2} \\
&= \left(g^{(1)} * g^{(2)} \right)(k_3) = g(k_3).
\end{aligned}$$

where the penultimate equality follows by Algorithm 3.6.

Finally, for every $x \notin I^j_k$, we have that $\mathbf{H}_{x,k} = 0$, i.e., property (c) holds for $\mathbf{H}$. The latter holds since $x \in I^j_{k_1}$ and $y \in I^j_{k_2}$ gives that $x + y \in I^{j+1}_{k_3(k_1,k_2,j)}$. Thus, by definition of approximate probability mass function, it follows that $g = g^{(1)} * g^{(2)}$ is an $(\epsilon, (j+1))$-approximation of $f = f^{(1)} * f^{(2)}$.

For the running time first recall that $L = O\left((c+d)\epsilon^{-1} \log n\right) = \tilde{O}(\epsilon^{-1})$. Since the cost for implementing Convolute is bounded by $\tilde{O}(L^2)$, it follows that we can implement this procedure in $\tilde{O}(\epsilon^{-2})$ time. □

The last ingredient we need is to show that given a family of probability mass functions, and their corresponding approximations, choosing one of these functions according to some probability distribution yields a random approximation in the natural way. Specifically, for an index set Q, let $\{f^{(q)}\}_{q\in Q}$ be a set of probability mass functions. Let $\hat{q}$ be be a random variable (independent from $\{f^{(q)}\}_{q\in Q}$) such that for every $q \in Q$, $\Pr[\hat{q} = q] = \mathbf{p}(q)$, and $\sum_{q\in Q} \mathbf{p}(q) = 1$. Furthermore, define

$$f := f^{(\hat{q})} = \sum_{q\in Q} \mathbf{p}(q) f^{(q)}$$

.

Lemma 3.6.3. *Suppose $g^{(q)}$ is an (ϵ, j)-approximation of the probability mass function f^q, for all $q \in Q$. Let f be the probability mass function as defined above. Then*

$$g := \sum_{q\in Q} \mathbf{p}(q) g^{(q)}$$

is an (ϵ, j)-approximation of f.

Proof. By definition of an (ϵ, j)-approximation, we know that there exist matrices $\{\mathbf{H}^{(q)}\}_{q\in Q}$ for $\{g^{(q)}\}_{q\in Q}$ satisfying properties (a), (b) and (c). We need to show that for g as defined in the lemma, there exist a suitable matrix $\mathbf{H}$ that satisifes each of these properties. To this end, define $\mathbf{H}$ as follows

$$\mathbf{H}_{x,k} := \sum_{q\in Q} \mathbf{p}(q) \mathbf{H}^{(q)}_{x,k}, \quad x \in \{0, \ldots, n^c\},\ k \in \{0, \ldots, L\}.$$

We start by showing property (a). Concretely, for any $z \in \{0, \ldots, n^c\}$ we get that

$$\sum_{k=0}^{L} \mathbf{H}_{x,k} = \sum_{k=0}^{L} \sum_{q\in Q} \mathbf{p}(q) \mathbf{H}^{(q)}_{x,k} = \sum_{q\in Q} \mathbf{p}(q) \sum_{k=0}^{L} \mathbf{H}^{(q)}_{x,k} = \sum_{q\in Q} \mathbf{p}(q) f^{(q)}(x) = f(x).$$

Next, $\mathbf{H}$ satisfies property (b) since for any $k \in \{0, \ldots, L\}$ we get that

$$\sum_{x=0}^{n^c} \mathbf{H}_{x,k} = \sum_{x=0}^{n^c} \sum_{q\in Q} \mathbf{p}(q) \mathbf{H}^{(q)}_{x,k} = \sum_{q\in Q} \mathbf{p}(q) \sum_{x=0}^{n^c} \mathbf{H}^{(q)}_{x,k} = \sum_{q\in Q} \mathbf{p}(q) g^{(q)}(k) = g(k).$$

Finally, for every $x \notin I^j_k$, we have that $\mathbf{H}_{x,k} = 0$, i.e., property (c) is satisfied for $\mathbf{H}$. The latter holds since for all $x \notin I^j_k$ we have that $\mathbf{H}^{(q)}_{x,k} = 0$ and thus $\mathbf{H}_{x,k} = \sum_{q\in Q} \mathbf{p}(q) \mathbf{H}^{(q)}_{x,k} = 0$. As a result we conclude that g is an (ϵ, j)-approximation of f with matrix $\mathbf{H}$ satisfying all the required properties. □

Algorithm 3.7: ComputeDistrib$(G, u, v, \ell, \epsilon)$

Input : Weighted graph $G = (V, E, \mathbf{w})$, with $\mathbf{w}(e) = [1, n^c]$ for each $e \in E$ and $c > 0$, two vertices $u, v \in V$, a length parameter $\ell \in [1, n^d]$ and an error parameter $\epsilon > 0$

Output : A vector $(j_{u,v}, g_\ell^{u,v}, p_\ell^{u,v})$, where $j_{u,v} \geq 1$ is a precision parameter, $g_\ell^{u,v}$ is an $(\epsilon, j_{u,v})$-approximation of $f_{s(w),\ell}^{u,v}$, and $p_\ell^{u,v}$ is the probability that the random walk w that starts at u hits v after ℓ steps

if $(\ell = 1)$ **then**
  If $(u, v) \notin E$, return $(1, \mathbf{0}, 0)$
  If $(u, v) \in E$, return $(1, g_\ell^{u,v}, p_\ell^{u,v})$, where $g_1^{u,v}(\frac{1}{\mathbf{w}(u,v)}) \leftarrow 1$ and $p_\ell^{u,v} \leftarrow \frac{\mathbf{w}(u,v)}{\mathbf{d}(v)}$
if $(\ell \geq 2)$ **then**
  Set $\ell' \leftarrow \lfloor \ell/2 \rfloor$ and $\ell'' \leftarrow \lceil \ell/2 \rceil$
  for *every* $y \in V$ **do**
    Invoke ComputeDistrib$(G, u, y, \ell', \epsilon)$ and ComputeDistrib$(G, y, v, \ell'', \epsilon)$
    Let $(j_{u,y}, g_{\ell'}^{u,y}, p_{\ell'}^{u,y})$ and $(j_{y,v}, g_{\ell''}^{y,v}, p_{\ell''}^{y,v})$ be the corresponding outputs
  Set $g_\ell^{u,v} \leftarrow \sum_{y \in V} p_{\ell'}^{u,y} p_{\ell''}^{y,v} \cdot (g_{\ell'}^{u,y} * g_{\ell''}^{y,v})$
  Return $\left((\max_{y \in V} \max(j_{u,y}, j_{y,v})) + 1, g_\ell^{u,v}, \sum_{y \in V} p_{\ell'}^{u,y} \cdot p_{\ell''}^{y,v} \right)$

We now describe how to compute a probability distribution that will in turn allow us to sample approximately from $f_{s(w),\ell}^{u,v}$. At a high level we accomplish this task by employing the "doubling technique" together with the approximate representations of the probability mass functions and their convolution. As an input, the algorithm receives a weighted graph G with polynomially bounded weights, a length parameter $\ell \geq 1$, an error parameter $\epsilon > 0$ and two vertices $u, v \in V$. The procedure computes and outputs a vector $\left(j_{u,v}, g_\ell^{u,v}, p_\ell^{u,v}\right)$, where $j_{u,v} \geq 1$ is a precision parameter, $g_\ell^{u,v}$ is an $(\epsilon, j_{u,v})$-approximation of $f_{s(w),\ell}^{u,v}$, and $p_\ell^{u,v} = \mathbb{P}_u [w_\ell = v]$ is the probability that the random walk w that originates at u hits v after ℓ steps.

If $(\ell = 1)$, then there are two possibilities depending on whether $(u, v) \in E$ or not. If the former holds, then the algorithms simply returns $(1, \mathbf{0}, 0)$ as it is not possible to reach v after performing one step of the random walk from v. Otherwise, we simply return $(1, g_1^{u,v}, p_\ell^{u,v})$, where $g_1^{u,v}(\frac{1}{\mathbf{w}(u,v)}) = 1$ and $p_\ell^{u,v} = \frac{\mathbf{w}(u,v)}{\mathbf{d}(v)}$.

However, if $(\ell > 1)$, then it first halves ℓ into two parts $\ell' = \lfloor \ell/2 \rfloor$ and $\ell'' = \lceil \ell/2 \rceil$. Next, for each $y \in V$ it recursively calls itself with input parameters $(G, u, y, \ell', \epsilon)$ and $(G, y, v, \ell'', \epsilon)$. The outputs from these two calls are then combined using the convolution manipulations described above to produce the final output. Exact details for implementing this procedure are summarized in Algorithm 3.7. The following lemma proves the correctness and the running time of the algorithm.

Lemma 3.6.4. *Given a weighted graph $G = (V, E, \mathbf{w})$ with $\mathbf{w}(e) \in [1, n^c]$ for each $e \in E$ and $c > 0$, two vertices $u, v \in V$, a length parameter $\ell \in [1, n^d]$ and an error parameter $\epsilon > 0$, ComputeDistrib$(G, u, v, \ell, \epsilon)$ (Algorithm 3.7) correctly computes a*

vector $(j_{u,v}, g_\ell^{u,v}, p_\ell^{u,v})$ *in* $\tilde{O}(n^3\epsilon^{-2})$ *time, where* $g_\ell^{u,v}$ *is an* $(\epsilon, j_{u,v})$*-approximation to* $f_{s(w),\ell}^{u,v}$ *and* $p_\ell^{u,v}$ *is the probability that the random walk* w *the starts at* u *hits* v *after* ℓ *steps. Moreover, the output* $j_{u,v}$ *cannot exceed* $O(\log n)$.

Proof. We first prove that the third coordinate of the output vector equals $\mathbb{P}_u[w_\ell = v]$. We proceed by induction on the length of the walk ℓ. If $(\ell = 1)$, it is easy to check that the condition holds by construction of the algorithm. Next assume $(\ell \geq 2)$ and note that $(\ell' < \ell)$ and $(\ell'' < \ell)$. Applying induction hypothesis on each recursion call, we know $p_{\ell'}^{u,y}$ is $\mathbb{P}_u[w_\ell = y]$ and $p_{\ell''}^{y,v}$ is $\mathbb{P}_y[w_{\ell''} = u]$. The latter along with the fact that $(\ell' + \ell'' = \ell)$ imply

$$\sum_{y \in V} \left(p_{\ell'}^{u,y} \cdot p_{\ell''}^{y,v}\right) = \sum_{y \in V} \left(\mathbb{P}_u[w_\ell = y] \cdot \mathbb{P}_y[w_{\ell''} = u]\right) = \mathbb{P}_u[w_\ell = v].$$

We next prove that the second coordinate $g_\ell^{u,v}$ is an (ϵ, j)-approximation of $f_{s(w),\ell}^{u,v}$. First, since $j_{u,v} = (\max_y \max(j_{v,y}, j_{y,u})) + 1$, Lemma 3.6.2 implies that $\left(g_{\ell'}^{u,y} * g_{\ell''}^{y,v}\right)$ is an $(\epsilon, j_{u,v})$-approximation of $f_{s(w),\ell',\ell'}^{u,y,v}$ where we define $f_{s(w),\ell',\ell'}^{u,y,v}$ to be the probability mass function of $s(w)$, conditioning on $w \sim w_{v,u}$, $\ell(w) = \ell' + \ell''$ and $w_{\ell'} = y$. Second, consider the triplets $\{(u, y, v)\}_{y \in V}$, and let $g_y = g_{\ell'}^{u,y} * g_{\ell''}^{y,v}$ and $p_y = p_{\ell'}^{u,y} \cdot p_{\ell'}^{y,v}$. Then by Lemma 3.6.3 we get that $g_\ell^{u,v} = \sum_{y \in V} p_y \cdot g_y$ is the desired $(\epsilon, j_{u,v})$-approximation.

Finally, we prove that $j_{u,v} = O(\log n)$. We will inductively show that the first coordinate $j_{u,v}$ of the output vector from ComputeDistrib$(G, u, v, \ell, \epsilon)$ is at most $k + 1$, for $\ell \leq 2^k$. For the base case $k = 0$, which implies that $\ell = 1$ and and the claim trivially holds. Now assume that $k \geq 1$. Since $\ell \leq 2^k$, by construction we get that $\ell' \leq 2^{k-1}$ and $\ell'' \leq 2^{k-1}$. By induction hypothesis, the first coordinates returned by all of the recursion calls are no more than $(k - 1) + 1 = k$. Thus, the returned $j_{u,v}$ at most $k + 1 = O(\log n)$.

For the running time, note that in all recursion calls of the procedure ComputeDistrib there are at most n^2 possible pairs (u, v) and $O(\log n)$ possible values of ℓ. In each of these calls, we invoke the procedure Convolute exactly n times, where each invocation costs $\tilde{O}(\epsilon^{-2})$ by Lemma 3.6.2. Thus the total running time is bounded by $\tilde{O}(n^3\epsilon^{-2})$.

□

We now have all the tools to prove Lemma 3.4.13.

Proof of Lemma 3.4.13. Our algorithm for sampling $s(w)$ is implemented as follows. First, it invokes the procedure computeDistrib$(G, u, v, \ell, \epsilon)$ and obtains the resulting vector $(j_{u,v}, g_\ell^{u,v}, p_\ell^{u,v})$. Then it samples from the distribution by choosing the interval $I_k^{j_{u,v}} = [\text{le}, \text{ri}]$ with probability $g_\ell^{u,v}(k)$, where $\text{le} := (1 + O(\frac{\epsilon}{\log n}))^k$ and $\text{ri} := (1 + O(\frac{\epsilon}{\log n}))^{k+j_{u,v}}$. Finally the algorithm outputs ri. This procedure is summarized in Algorithm 3.8.

Algorithm 3.8: SAMPLE$(G, u, v, \ell, \epsilon)$

Input : Weighted graph $G = (V, E, \mathbf{w})$, with $\mathbf{w}(e) = [1, n^c]$ for each $e \in E$ and some $c > 0$, two vertices $u, v \in V$, a length parameter $\ell \in [1, n^d]$ and an error parameter $\epsilon > 0$

Output : A sampled $s(w)$ up to a $(1+\epsilon)$ relative error, where w is a random walk of length ℓ that starts at u and ends at v

1 Set $(j_{u,v}, g_\ell^{u,v}, p_\ell^{u,v}) \leftarrow$ COMPUTEDISTRIB$(G, u, v, \ell, O(\frac{\epsilon}{\log n}))$

2 Let k_0 be the index of the interval $I_{k_0}^{j_{u,v}}$ that is sampled according to distribution $g_\ell^{u,v}$

3 Return $\left(1 + O(\frac{\epsilon}{\log n})\right)^{k_0 + j_{u,v}}$

We next argue about the correctness. Note that by property (b) in the definition of approximation $g_\ell^{u,v}$ of $f_{s(w),\ell}^{u,v}$, this sampling process can be viewed as sampling the pair (x, i) from the distribution $\mathbf{H}_{x,i}$, without knowing x. Furthermore, by property (a), each x is sampled with the correct probability $f_{s(w),\ell}^{u,v}(x)$. Since we can restrict to $\epsilon \leq 1/2$ it follows by Lemma 3.6.4 that ri/le $= (1+O(\frac{\epsilon}{\log n}))^{j_{u,v}} \leq (1+\epsilon)$. Thus by property (c) we get that ri is within $[x, (1+\epsilon)x]$ for the (unknown) sampled x.

The running time of our sampling procedure is asymptotically dominated by the running time of COMPUTEDISTRIB, which is in turn bounded by $O(n^3\epsilon^{-2})$, as desired. □

3.7 Schur Complement Sparsifier from Sum of Random Walks

In this section we prove Theorem 3.3.1, which states that sampling random walks generates sparsifiers of Schur complements:

Lemma 3.7.1 (Restatement of Theorem 3.3.1)**.** *Let $G = (V, E, w)$ be an undirected, weighted multi-graph with a subset of vertices K. Furthermore, let $\epsilon \in (0, 1)$, and let ρ be some parameter related to the concentration of sampling given by*

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

Let H be an initially empty graph, and for every edge $e = (u, v)$ of repeat ρ times the following procedure:

1. *Simulate a random walk starting from u until it first hits K at vertex t_1,*
2. *Simulate a random walk starting from v until it first hits K at vertex t_2,*
3. *Combine these two walks (including e) to get a walk $u = (t_1 = u_0, \ldots, u_\ell = t_2)$, where ℓ is the length of the combined walk.*

4. *Add the edge* (t_1, t_2) *to* H *with weight*

$$1/\left(\rho \sum_{i=0}^{\ell-1} (1/\mathbf{w}(u_i, u_{i+1}))\right)$$

The resulting graph H *satisfies* $\mathbf{L}_H \approx_\epsilon \mathbf{SC}(G, K)$ *with high probability.*

Note that this rescaling by $1/\left(\rho \sum_{i=0}^{\ell-1} (1/\mathbf{w}(u_i, u_{i+1}))\right)$ is quite natural: Consider the degenerate case where $K = V$. This routine generates ρ copies of each edge weight, which then need to be rescaled by $1/\rho$ to ensure approximation to the original graph.

Similar to other randomized graph sparsification algorithms [12, 89, 145, 167, 235], our sampling scheme directly interacts with Chernoff bounds. Our random matrices are 'groups' of edges related to random walks starting from the edge e. We will utilize Theorem 1.1 due to [252], which we paraphrase in our notion of approximations.

Theorem 3.7.2. *Let* $\mathbf{X}_1, \mathbf{X}_2 \ldots \mathbf{X}_k$ *be a set of random matrices satisfying the following properties:*

1. *Their expected sum is a projection operator onto some subspace, i.e.,* $\sum_i \mathbb{E}[\mathbf{X}_i] = \mathbf{\Pi}$.
2. *For each* $\mathbf{X}_i$*, its entire support satisfies:* $0 \preceq \mathbf{X}_i \preceq \frac{\epsilon^2}{O(\log n)}\mathbf{I}$.

Then, with high probability, we have

$$\sum_i \mathbf{X}_i \approx_\epsilon \mathbf{\Pi}.$$

Re-normalizations of these bounds similar to the work of [235] give the following graph theoretic interpretation of the theorem above.

Corollary 3.7.3. *Let* $E_1 \ldots E_k$ *be distributions over random edges satisfying the following properties:*

1. *Their expectation sums to the graph* G*, i.e.,* $\sum_i \mathbb{E}[E_i] = G$.
2. *For each* E_i*, any edge in its support has low leverage score in* G*, i.e.,* $\mathbf{w}(e) R^G_{\text{eff}}(e) \leq \frac{\epsilon^2}{O(\log n)}$.

Then, with high probability, we have

$$\sum_i \mathbf{L}_{E_i} \approx_\epsilon \mathbf{L}_G.$$

To fit the sampling scheme outlined in Theorem 3.3.1 into the requirements of Corollary 3.7.3, we need (1) a specific interpretation of Schur complements in terms of walks, and (2) a bound on the effective resistances between two vertices at a given distance.

Given a walk $w = u_0, \ldots, u_\ell$ of length ℓ in G with a subset a vertices K, we say that w is a *terminal-free* walk iff $u_0, u_\ell \in K$ and $u_1, \ldots, u_{\ell-1} \in V \setminus K$.

Fact 3.7.4 ([89], Lemma 5.4)**.** *For any undirected, unweighted graph G and any subset of vertices $K \subseteq V$, the Schur complement $\mathbf{SC}(G, K)$ is given as an union over all multi-edges corresponding to terminal-free walks $u_0, \ldots, u_\ell$ with weight*

$$\frac{\prod_{i=0}^{\ell-1} \mathbf{w}(u_i, u_{i+1})}{\prod_{i=1}^{\ell-1} \mathbf{d}(u_i)}.$$

The fact below follows by repeatedly applying the triangle inequality of the effective resistances between two vertices.

Fact 3.7.5. *In an weighted undirected graph G, the effective resistance between two vertices that are connected by a path $p = (p_0, \ldots, p_\ell)$ is at most $\sum_{i=0}^{\ell-1} 1/\mathbf{w}(p_i, p_{i+1})$.*

Combining the above results gives the guarantees of our sparsification routine.

Proof of Theorem 3.3.1. For every edge $e \in E$, let W_e be the random graph corresponding the the terminal-free random walk that started at edge e. Define $H = \rho \cdot \sum_e W_e$ to be the output graph by our sparsification routine, where $\rho = O(\log n \epsilon^{-2})$ is the sampling overhead. To prove that $\mathbf{L}_H \approx_\epsilon \mathbf{SC}(G, K)$ with high probability, we need to show that (1) $\mathbb{E}[H] = \mathbf{SC}(G, K)$ and (2) for any edge f in W_e, its leverage score $\mathbf{w}(f) R_{\text{eff}}^{W_e}(f)$ is at most $\epsilon^2 / \log n$ (by Corollary 3.7.3). Note that (2) immediately follows from the effective resistance bound of Fact 3.7.5 and the choice of $\rho = O(\log n / \epsilon^2)$. We next show (1).

To this end, we start by describing the decomposition of $\mathbf{SC}(G, K)$ into random multi-edges, which correspond to random terminal-free walks in Fact 3.7.4. The main idea is to sub-divide each walk $u_0 \ldots u_\ell$ of length ℓ in G into ℓ walks of the same length, each starting at one of the ℓ edges on the walk, and each having weight

$$\frac{\prod_{i=0}^{\ell-1} \mathbf{w}(u_i, u_{i+1})}{\prod_{i=1}^{\ell-1} \mathbf{d}(u_i)}$$

By construction of our sparsification routine, note that every random graph W_e is a distribution over walks $u_0 \ldots u_\ell$, each picked with probability

$$\frac{1}{\mathbf{w}(e)} \frac{\prod_{i=0}^{\ell-1} \mathbf{w}(u_i, u_{i+1})}{\prod_{i=1}^{\ell-1} \mathbf{d}(u_i)}.$$

Thus, to retain expectation, when such a walk is picked, our routine correctly adds it to H with weight $1/(\rho \sum_{i=0}^{\ell-1} 1/\mathbf{w}(u_i, u_{i+1}))$.

Formally, we get the following chain of equalities

$$\begin{aligned}
&\mathbb{E}\left[H\right] = \rho \cdot \sum_{e} \mathbb{E}\left[W_e\right] \\
&= \rho \cdot \sum_{e} \sum_{w=u_0,u_1\ldots u_{\ell(w)}:w\ni e} \frac{1}{\rho\left(\sum_{i=0}^{\ell-1} 1/\mathbf{w}(u_i,u_{i+1})\right)} \cdot \frac{1}{\mathbf{w}(e)} \cdot \frac{\prod_{i=0}^{\ell-1}\mathbf{w}(u_i,u_{i+1})}{\prod_{i=1}^{\ell-1}\mathbf{d}\left(u_i\right)} \\
&= \sum_{w=u_0,u_1\ldots u_{\ell(w)}} \sum_{e:e\in w} \frac{1}{\left(\sum_{i=0}^{\ell-1} 1/\mathbf{w}(u_i,u_{i+1})\right)} \cdot \frac{1}{\mathbf{w}(e)} \cdot \frac{\prod_{i=0}^{\ell-1}\mathbf{w}(u_i,u_{i+1})}{\prod_{i=1}^{\ell-1}\mathbf{d}\left(u_i\right)} \\
&= \sum_{w=u_0,u_1\ldots u_{\ell(w)}} \frac{\prod_{i=0}^{\ell-1}\mathbf{w}(u_i,u_{i+1})}{\prod_{i=1}^{\ell-1}\mathbf{d}\left(u_i\right)} \\
&= \mathbf{SC}(G,K).
\end{aligned}$$

□

3.8 Conclusion

In this chapter, we study algorithms for dynamically maintaining all-pairs effective resistances in undirected weighted graphs and Laplacian solvers in undirected, unweighted, bounded degree graphs. In particular, we obtain an algorithm with $O(m^{3/4}\epsilon^{-4})$ update and query time for $(1+\epsilon)$-approximating effective resistances in unweighted graphs, and an algorithm with $O(n^{11/12}\epsilon^{-5})$ update and query time for solving Laplacian systems approximately while allowing implicit access to few entries of the solution vector. Our key component is the dynamic maintenance of spectral vertex sparsifiers (also known as approximate Schur complements) with respect to a set of terminals of our choice.

A natural attempt to improve the running times of our effective resistance data-structure is to employ a hierarchy of dynamic spectral vertex sparsifiers. However, this is not an easy task as there are many dependencies which one needs to deal with when employing a hierarchical approach. We believe that a careful analysis combined with a way to control the propagation of updates among levels might indeed lead to further improvements.

Our dynamic Schur complement data-structure works only against an oblivious adversary. While this is a standard assumption in dynamic algorithms, especially when designing the first non-trivial algorithm for a particular problem, it is highly desirable to remove this assumption as this might lead to other algorithmic applications. A good starting step would be to design a randomized algorithm that works against an adaptive adversary.

Perhaps one of the most important problems is to remove our bounded-degree assumption for dynamic Laplacian solvers with demand vectors that have large non-zero support. Our current algorithm exploits this assumption in several places, the most critical one being the bound on the load on any vertex induced by the random

walks that our algorithm maintains. This suggests that new approaches might be required to be able to remove this assumption.

CHAPTER 4

Dynamic Low-Stretch Trees via Dynamic Low-Diameter Decompositions

Spanning trees of low average stretch on the non-tree edges, as introduced by Alon et al. [17], are a natural graph-theoretic object. In recent years, they have found significant applications in solvers for symmetric diagonally dominant (SDD) linear systems. In this work, we provide the first dynamic algorithm for maintaining such trees under edge insertions and deletions to the input graph. Our algorithm has update time $n^{1/2+o(1)}$ and the average stretch of the maintained tree is $n^{o(1)}$, which matches the stretch in the seminal result of Alon et al.

Similar to Alon et al., our dynamic low-stretch tree algorithm employs a dynamic hierarchy of low-diameter decompositions (LDDs). As a major building block we use a dynamic LDD that we obtain by adapting the random-shift clustering of Miller et al. [195] to the dynamic setting. The major technical challenge in our approach is to control the propagation of updates within our hierarchy of LDDs: each update to one level of the hierarchy could potentially induce several insertions *and* deletions to the next level of the hierarchy. We achieve this goal by a sophisticated amortization approach. In particular, we give a bound on the number of changes made to the LDD per update to the input graph that is significantly better than the trivial bound implied by the update time.

We believe that the dynamic random-shift clustering might be useful for independent applications. One of these applications is the dynamic spanner problem. By combining the random-shift clustering with the recent spanner construction of Elkin and Neiman [92]. We obtain a fully dynamic algorithm for maintaining a spanner of stretch $2k-1$ and size $O(n^{1+1/k}\log n)$ with amortized update time $O(k\log^2 n)$ for any integer $2 \leq k \leq \log n$. Compared to the state-of-the art in this

regime Baswana et al. [33], we improve upon the size of the spanner and the update time by a factor of k.

4.1 Introduction

Graph compression is an important paradigm in modern algorithm design. Given a graph G with n nodes, can we find a substantially smaller (read: sparser) subgraph H such that H preserves central properties of G? Very often, this compression is "lossy" in the sense that the properties of interest are only preserved approximately. A ubiquitous example of graph compression schemes are *spanners*: every graph G admits a spanner H with $O(n^{1+1/k})$ edges that has *stretch* $2k-1$ (for any integer $k \geq 2$), meaning that for every edge $e = (u, v)$ of G not present in H there is a path from u to v in H of length at most $2k-1$. Thus, when $k = \log n$, very succinct compression with $O(n)$ edges can be achieved at the price of stretch $O(\log n)$.

The most succinct form of subgraph compression is achieved when H is a tree. Spanning trees, for example, are a well-known tool for preserving the connectivity of a graph. It is thus natural to ask whether, similar to spanners, one could also have spanning trees with low stretch for each edge. This unfortunately is known to be false: in a ring of n nodes every tree will result in a stretch of $n-1$ for the single edge not contained in the tree. However, it turns out that a quite similar goal can be achieved by relaxing the concept of stretch: every graph G admits a spanning tree T of *average stretch* $O(\log n \log \log n)$ [13], where the average stretch is the sum of the stretches of all edges divided by the total number of edges. Such subgraphs are called *low (average) stretch trees* and have found numerous applications in recent years, most notably in the design of fast solvers for symmetric diagonally dominant (SDD) linear systems [51, 74, 161, 168, 169, 237]. We believe that their fundamental graph-theoretic motivation and their powerful applications make low-stretch trees a very natural object to study as well in a dynamic setting, similar to spanners [27, 33, 54, 90] and minimum spanning trees [96, 106, 128, 138, 203, 260]. Indeed, the design of a dynamic algorithm for maintaining a low-stretch tree was posed as an open problem by Baswana et al. [33], but despite extensive research on dynamic algorithms in recent years, no such algorithm has yet been found.

In this chapter, we give the first non-trivial algorithm for this problem in the *dynamic* setting. Specifically, we maintain a low-stretch tree T of a dynamic graph G undergoing updates in the form of edge insertions and deletions in the sense that after each update to G we compute the set of necessary changes to T. The goal in this problem is to keep the time spent after each update small while still keeping the average stretch of T tolerable. Our main result is a fully dynamic algorithm for maintaining a spanning tree of expected average stretch $n^{o(1)}$ with expected amortized update time $n^{1/2+o(1)}$. At a high level, we obtain this result by combining the classic low-stretch tree construction of Alon et al. [17] with a dynamic algorithm for maintaining low diameter decompositions (LDD) based on random-shift clustering [195]. Our LDD algorithm might be of independent interest, and we provide

another application by using it to obtain a dynamic version of the recent spanner construction of Elkin and Neiman [92]. The resulting dynamic spanner algorithm improves upon one of the state-of-the-art algorithms by Baswana et al. [33].

Our overall approach towards the low-stretch tree algorithm – to use low-diameter decompositions based on random-shift clustering in the construction of Alon et al. [17] – has been used before in parallel and distributed algorithms [51, 111, 124]. However, to make this approach work in the dynamic setting we need to circumvent some non-trivial challenges. In particular, we cannot employ the following paradigm that often is very helpful in designing dynamic algorithms: design an algorithm that can only handle edge deletions and then extend it to the fully dynamic setting using a general reduction. While we do follow this paradigm for our dynamic LDD algorithm, there are two obstacles that prevent us from doing so for the dynamic low-stretch tree: First, many fully-dynamic-to-decremental reductions exploit some form of "decomposability", which does not hold for low-stretch trees, i.e., low-stretch trees of subgraphs of the input graph cannot be simply be combined to a single low-stretch tree of the full graph. Second, in our dynamic low-diameter decomposition edges might start and stop being inter-cluster edges, even if the input graph is only undergoing deletions. In the hierarchy of Alon et al. this leads to both insertions and deletions at the next level of the hierarchy. As opposed to other dynamic problems [12, 134], one algorithm cannot simply enforce some type of "monotonicity" by not passing on insertions to the next level of the hierarchy (to stay within a deletions-only setting) as there might be too many such edges to ignore them. Thus, it seems that we really have to deal with the fully dynamic setting in the first place. We show that this can be done by a sophisticated amortization approach that explicitly analyzes the number of updates passed on to the next level.

Related Work. Low average stretch trees have been introduced by Alon et al. [17] who obtained an average stretch of $2^{O(\sqrt{\log n \log\log n})}$ and also gave a lower bound of $\Omega(\log n)$ on the average stretch. The first construction with polylogarithmic average stretch was given by Elkin et al. [91]. Further improvements [8, 168] culminated in the state-of-the-art construction of Abraham and Neiman [13] with average stretch $O(\log n \log\log n)$. All these trees with polylogarithmic average stretch can be computed in time $\tilde{O}(m)$. To the best of our knowledge, all known algorithms in parallel and distributed models of computation [51, 111, 124] are based on the scheme of Alon et al. and thus do not provide polylogarithmic stretch guarantees.

The main application of low-stretch trees has been in solving symmetric, diagonally dominant (SDD) systems of linear equations. It has been observed that iterative methods for solving these systems can be made faster by preconditioning with a low-stretch tree [55, 239, 253]. Consequently, they have been an important ingredient in the breakthrough result of Spielman and Teng [237] for solving SDD systems in nearly linear time. In this solver, low-stretch trees are utilized for constructing ultra-sparsifiers, which in turn are used as preconditioners. Beyond this initial breakthrough, low-stretch trees have also been used in subsequent,

faster solvers [51, 74, 161, 168, 169]. Another prominent application of low-stretch trees (concretely, the variant of random spanning trees with low expected stretch) is the remarkable cut-based graph decomposition of Räcke [23, 214], which embeds any general undirected graph into convex combination of spanning trees, while paying only a $\tilde{O}(\log n)$ congestion for the embedding. This decomposition tool, initially aimed at giving the best competitive ratio for oblivious routing, has found several applications ranging from approximation algorithms for cut-based problems (e.g., minimum bisection [214]) to graph compression (e.g., vertex sparsifiers [197]). Other classic problems in the realm of approximation algorithms that utilize the properties of low-stretch trees include the k-server problem [17] and the minimum communication cost spanning tree problem [140, 207].

In terms of dynamic algorithms, we are not aware of any prior work for maintaining low-stretch trees. The closest related works are arguably dynamic algorithms for maintaining distance oracles and spanners, as they also aim preserving pairwise distances, and dynamic algorithms for maintaining minimum spanning trees, as they also are spanning trees with an additional property.

A distance oracle is a data structure that can answer queries for the (approximate) distance between a pair of nodes. The fully dynamic distance oracle of Abraham, Chechik, and Talwar [11] for unweighted, undirected graphs has expected amortized update time $\tilde{O}(\sqrt{m}n^{1/k})$, query time $O(k^2\rho^2)$, and stretch $2^{O(k\rho)}$, where the parameter $k \geq 2$ is integer and $\rho = 1+\lceil \frac{\log n^{1-1/k}}{\log(m/n^{1-1/k})} \rceil$. To the best of our knowledge, the recent decremental distance oracle of Chechik [61] can be used to extend this result to weighted graphs and to improve the stretch and the query time, while leaving the update time essentially unchanged.

For dynamic spanner algorithms, the main goal is to maintain, for any given integer $k \geq 2$, a spanner of stretch $2k-1$ with $\tilde{O}(n^{1+1/k})$ edges. Spanners of stretch $2k-1$ and size $O(n^{1+1/k})$ exist for every undirected graph [28], and this trade-off is presumably tight under Erdős's girth conjecture. The dynamic spanner problem has been introduced by Ausiello et al. [27]. They showed how to maintain a 3- or 5-spanner with amortized update time proportional to the maximum degree of the graph. Using techniques from the streaming literature, Elkin [90] provided an algorithm for maintaining a $(2k-1)$-spanner with $\tilde{O}(mn^{-1/k})$ expected update time. Faster update times were achieved by Baswana et al. [33]: their algorithms maintain $(2k-1)$-spanners either with expected amortized update time $O(1)^k$ or with expected amortized update time $O(k^2 \log^2 n)$. Later, Bodwin and Krinninger [54] initiated the study of dynamic spanners with worst-case update times, and recently, Bernstein, Forster, and Henzinger [45] presented a deamortization approach to maintain $(2k-1)$-spanners with high-probability worst-case update time $O(1)^k \log^3 n$. All of these algorithms exhibit the stretch/space trade-off mentioned above in unweighted graphs, up to polylogarithmic factors in the size of the spanner.

The first non-trivial algorithm for dynamically maintaining a minimum spanning tree was developed by Frederickson [106] and had a worst-case update time

of $O(\sqrt{m})$. Using a general sparsification technique, this bound was improved to $O(\sqrt{n})$ by Eppstein et al. [96]. In terms of amortized bounds, Holm et al. [138] were the first to improve this bound and obtained polylogarithmic amortized update time. A recent breakthrough of Nanongkai, Saranurak, and Wulff-Nilsen [202, 203, 260], who finally achieved a *worst-case* update time of $n^{o(1)}$.

Our Results. Our main result is a dynamic algorithm for maintaining a low average stretch tree of an unweighted, undirected graph.

Theorem 4.1.1. *Given any unweighted, undirected graph undergoing edge insertions and deletions, there is a fully dynamic algorithm for maintaining a spanning forest of expected average stretch $n^{o(1)}$ that has expected amortized update time $m^{1/2+o(1)}$. These guarantees hold against an oblivious adversary.*

This is the first non-trivial algorithm for this fundamental problem. Our stretch matches the seminal construction of Alon et al. [17], which is still the state of the art in parallel and distributed settings [51, 111, 124].

Similar to the approach of [167] in the *static* setting, we can apply Theorem 4.1.1 to a cut sparsifier of the input graph, which has only $\tilde{O}(n)$ edges, to improve the running time for dense graphs. Such a cut sparsifier can be maintained with the dynamic algorithm of Abraham et al. [12] that has polylogarithmic update time.

Corollary 4.1.2. *Given any unweighted, undirected graph undergoing edge insertions and deletions, there is a fully dynamic algorithm for maintaining a spanning forest of expected average stretch $n^{o(1)}$ that has expected amortized update time $n^{1/2+o(1)}$. These guarantees hold against an oblivious adversary.*

Obtaining this improvement is non-trivial because cut sparsifiers are weighted graphs, even when the input graph is unweighted, and the algorithm of Theorem 4.1.1 only accepts unweighted graphs. To deal with this issue, we deviate from the approach of [167] by interpreting the edge weights of the sparsifier as edge multiplicities in an unweighted graph. A fine-grained analysis of the amount of change to edge the multiplicities per update to the input graph then allows us to get the desired benefits of combining both algorithms.

We additionally show that $\sqrt{n}$ is not an inherent barrier to the update time, at least if very large stretch is tolerated. A modification of our algorithm gives average stretch $O(t)$ and update time $\frac{n^{1+o(1)}}{t}$ for $t \geq \sqrt{n}$.

One of the main building blocks of our dynamic low-stretch tree algorithm is a dynamic algorithm for maintaining a low-diameter decomposition (LDD). Roughly speaking, for $\beta \in (0, 1)$ and $\Delta > 0$, a (β, Δ)-decomposition of a graph is a partitioning of its nodes into node-disjoint clusters such that (1) any pair of nodes belonging to the same cluster are at distance at most Δ, and (2) the number of edges whose endpoints belong to different clusters is bounded by βm. The following theorem gives a dynamic variant of such decompositions.

Theorem 4.1.3. *Given any unweighted, undirected multigraph undergoing edge insertions and deletions, there is a fully dynamic algorithm for maintaining a $(\beta, O(\frac{\log n}{\beta}))$-decomposition (with clusters of strong diameter $O(\frac{\log n}{\beta})$ and at most βm inter-cluster edges in expectation) that has expected amortized update time $O(\log^2 n/\beta^2)$. A spanning tree of diameter $O(\frac{\log n}{\beta})$ for each cluster can be maintained in the same time bound. The expected amortized number of edges to become inter-cluster edges after each update is $O(\log^2 n/\beta)$. These guarantees hold against an oblivious adversary.*

Our algorithm is based on the random-shift clustering of Miller at al. [195], with many tweaks to make it work in a dynamic setting. In our analysis of the algorithm, we bound the amortized number of changes to the clustering per update by $\tilde{O}(1/\beta)$, which is significantly smaller than the naive bound of $\tilde{O}(1/\beta^2)$ implied by the update time. This is particularly important for hierarchical approaches, such as in our dynamic low-stretch tree algorithm, because a small bound on the number of amortized changes helps in controlling the number of induced updates to be processed within the hierarchy. Independently, Saranurak and Wang [225] obtained a fully dynamic LLD algorithm with nearly the same guarantees (up to polylogarithmic factors).[1] We believe that our solution is arguably simpler than their expander pruning approach.

The dynamic random-shift clustering underlying our dynamic LDD is of independent interest. A direct consequence demonstrating the usefulness of our dynamic random-shift clustering algorithm is the following new result for the dynamic spanner problem.

Theorem 4.1.4. *Given any unweighted, undirected graph undergoing edge insertions and deletions, there is a fully dynamic algorithm for maintaining a spanner of stretch $2k-1$ and expected size $O(n^{1+1/k}\log n)$ that has expected amortized update time $O(k\log^2 n)$. These guarantees hold against an oblivious adversary.*

Recall that the fully dynamic algorithm of Baswana et al. [33] maintains a spanner of stretch $2k-1$ and expected size $O(kn^{1+1/k}\log n)$ with expected amortized update time $O(k^2\log^2 n)$. Our new algorithm thus improves both the size and the update time by a factor of k. This is particularly relevant because the stretch/size trade-off of $2k-1$ vs. $O(n^{1+1/k})$ is tight under the girth conjecture. We thus exceed the conjectured optimal size by a factor of only $\log n$ compared to the prior $k\log n$, where k might be as large as $\log n$. When we restrict ourselves to the decremental setting, we do achieve size $O(n^{1+1/k})$ with expected amortized update time $O(k\log n)$. Again, this saves a factor of k compared to Baswana et al. [33]. To obtain Theorem 4.1.4, we employ our dynamic random-shift clustering algorithm in the spanner construction of Elkin and Neiman [92] and combine it with the dynamic spanner framework of Baswana et al. [33].

[1] The low-diameter decomposition of Saranurak and Wang can be maintained against an adaptive online adversary. However, the low-diameter spanning trees of their clustering can only be maintained against an oblivious adversary. Therefore, plugging in their dynamic LDD algorithm into our dynamic low-stretch tree construction does not yield any improvement over our guarantees.

Structure of this Chapter. The remainder of this chapter is structured as follows. We first settle the notation and terminology in Section 4.2. We then give a high-level overview of our results and techniques in Section 4.3. Finally, we provide all necessary details for our dynamic low-stretch tree (Section 4.4), our dynamic low-diameter decomposition (Section 4.5), and our dynamic spanner algorithm (Section 4.6).

4.2 Preliminaries

Graphs. Let $G = (V, E, \mathbf{w}_G)$ be an undirected weighted graph, where $n = |V|$, $m = |E|$ and $\mathbf{w}_G : E \to \mathbb{R}_+$. If $\mathbf{w}_G(e) = 1$ for all $e \in E$, then we say G is an undirected unweighted graph. If E is a multiset, i.e., every element of E may have integer multiplicity greater than 1, then we call G a multigraph. For a subset $C \subseteq V$ let $G[C]$ denote the subgraph of G induced by C. Throughout the chapter we call $C \subset V$ a *cluster*. For any positive integer k, a *clustering* of G is a partition of V into disjoint subsets $C_1, C_2, \ldots, C_k$. We say that an edge is an *intra-cluster* edges if both its endpoints belong to the same cluster C_i for some i; otherwise, we say that an edge is an *inter-cluster* edge.

For any $u, v \in V$ let $\text{dist}_G(u, v)$ denote the length of a shortest path between u and v induced by the edge weights $\mathbf{w}_G$ of the graph G. When G is clear from the context, we will omit the subscript. The *strong diameter* of a cluster $C \subset V$ is the maximum length of the shortest path between two nodes in $G[C]$, i.e., $\max\{\text{dist}_{G[C]}(u, v) \mid u, v \in C\}$. In the following we define a low-diameter clustering of G.

Definition 4.2.1. *Let k be any positive integer, $\beta \in (0, 1)$ and $\Delta > 0$. Given an undirected, unweighted graph $G = (V, E)$, a (β, Δ)-decomposition of G is a partition of V into disjoint subsets $C_1, C_2, \ldots, C_k$ such that:*

1. *The strong diameter of each C_i is at most Δ.*
2. *The number of edges with endpoints belonging to different subsets is at most βm.*

In the (β, Δ)-decompositions of the randomized dynamic algorithms in this chapter, the bound in Condition 2 is in expectation.

Let $H = (V, F)$ be a subgraph of $G = (V, E, \mathbf{w}_G)$. For any pair of nodes $u, v \in V$, we let $\text{dist}_H(u, v)$ denote the length of a shortest path between u and v in H. We define the *stretch* of an edge $(u, v) \in E$ with respect to H to be

$$\text{stretch}_H(u, v) \coloneqq \frac{\text{dist}_H(u, v)}{\mathbf{w}_G(u, v)}.$$

The stretch of H is defined as the maximum stretch of any of edge $(u, v) \in E$. The *average stretch* over all edges of G with respect to H is given by

$$\text{avg-stretch}_H(G) \coloneqq \frac{1}{|E|} \sum_{(u,v)\in E} \text{stretch}_H(u, v).$$

Exponential Distribution. For a parameter λ, the probability density function of the *exponential distribution* $\mathrm{Exp}(\lambda)$ is given by

$$f(x,\lambda) := \begin{cases} \lambda e^{-\lambda x} & \text{if } x \geq 0 \\ 0 & \text{otherwise.} \end{cases}$$

The mean of the exponential distribution is $1/\lambda$.

Dynamic Algorithms. Consider a graph with n nodes undergoing updates in the form of edge insertions and edge deletions. An *incremental* algorithm is a dynamic algorithm that can only handle insertions, a *decremental* algorithm can only handle deletions, and a *fully dynamic* algorithm can handle both. We follow the convention that a fully dynamic algorithm starts from an empty graph with n nodes. The (maximum) running time spent by a dynamic algorithm for processing each update (before the next update arrives) is called *update time*. We say that a dynamic algorithm has *(expected) amortized* update time $u(n)$ if its total running time spent for processing a sequence of q updates is bounded by $q \cdot u(n)$ (in expectation). In this chapter, we assume that the updates to the graph are performed by an *oblivious adversary* who fixes the sequences of updates in advance, i.e., the adversary is not allowed to adapt its sequence of updates as the algorithm proceeds. This is a standard assumption in dynamic graph algorithms[2] and in particular, it implies that for randomized dynamic algorithms the sequence of updates is independent from the random choices of the algorithm.

4.3 Technical Overview

In the following, we provide some intuition for our approach and highlight the main ideas of this chapter.

Low Average Stretch Tree. A first idea is to employ the dynamic low-diameter decomposition of Theorem 4.1.3. This algorithm can maintain a $(\beta, O(\frac{\log n}{\beta}))$-decomposition, i.e., a partitioning of the graph into clusters such that there are at most βm inter-cluster edges and the (strong) diameter of each cluster is at most $O(\frac{\log n}{\beta})$. In particular, each cluster has a designated center and the algorithm maintains a spanning tree of each cluster in which every node is at distance at most $O(\frac{\log n}{\beta})$ from the center. Now consider the following simple dynamic algorithm:

1. Maintain a $(\beta, O(\frac{\log n}{\beta}))$-decomposition of the input graph G.
2. Contract the clusters in the decomposition to single nodes and maintain a multigraph G' containing one node for each cluster and all inter-cluster edges.

[2]For example, all known randomized dynamic spanner algorithms [33, 45, 54, 90] work under this assumption.

3. Compute a low-stretch tree T' of G' after each update to G using a static algorithm providing polylogarithmic average stretch.

4. Maintain T as the "expansion" of T' in which every node in T' is replaced by the spanning tree of diameter $O(\frac{\log n}{\beta})$ of the cluster representing the node.

As the clusters are non-overlapping it is immediate that T is indeed a tree. To analyze the average stretch of T, we distinguish between inter-cluster edges (with endpoints in different clusters) and intra-cluster edges (with endpoints in the same cluster). Each intra-cluster edge has stretch at most $O(\frac{\log n}{\beta})$ as the spanning tree of the cluster containing both endpoints of such an edge is a subtree of T. Each inter-cluster edge has polylogarithmic average stretch in T' with respect to G'. By expanding the clusters, the length of each path in T' increases by a factor of at most $O(\frac{\log n}{\beta})$. Thus, inter-cluster edges have an average stretch of $O(\frac{\log n}{\beta} \operatorname{polylog} n)$ in T. As there are at most m intra-cluster edges and at most βm inter-cluster edges, the total stretch over all edges is at most $O(m \cdot \frac{\log n}{\beta} + \beta m \cdot \frac{\log n}{\beta} \operatorname{polylog} n) = \tilde{O}(m \cdot \frac{1}{\beta})$, which gives an average stretch of $\tilde{O}(\frac{1}{\beta})$.

To bound the update time, first observe that the number of inter-cluster edges is at most βm. Thus, G' has at most βm edges and therefore the static algorithm for computing T' takes time $\tilde{O}(\beta m)$ per update. Together with the update time of the dynamic LDD, we obtain an update time of $\tilde{O}(\frac{1}{\beta^2} + \beta m)$. By setting $\beta = m^{1/3}$, we would already obtain an algorithm for maintaining a tree of average stretch $\tilde{O}(m^{1/3})$ with update time $\tilde{O}(m^{2/3})$.

We can improve the stretch and still keep the update time sublinear by a hierarchical approach in which the scheme of clustering and contracting is repeated k times. Observe that the i-th contracted graph will contain at most $\beta^i m$ many edges and, in the final tree T, the stretch of each edge disappearing with the $(i+1)$-th contraction is $O(\frac{\log n}{\beta})^{i+1}$, which can be obtained by expanding the contracted low-diameter clusters. After k contractions, there are at most $\beta^k m$ edges remaining and they have polylogarithmic average stretch in T' with respect to G', which, again by expanding clusters, implies an average stretch of at most $O(\frac{\log n}{\beta})^k \cdot \operatorname{polylog} n$ in T with respect to G. This leads to a total stretch of $O(\sum_{0 \le i \le k-1} \beta^i m \cdot O(\frac{\log n}{\beta})^{i+1} + \beta^k m \cdot O(\frac{\log n}{\beta})^k \operatorname{polylog} n) = \tilde{O}(m \cdot \frac{O(\log n)^k}{\beta})$, which gives an average stretch of $\tilde{O}(\frac{O(\log n)^k}{\beta})$. To bound the update time, observe that updates propagate within the hierarchy as each change to inter-cluster edges of one layer will appear as an update in the next layer. Each operation in the dynamic LDD algorithm will perform at most one change to the clustering, i.e., the number of changes propagated to the next layer of the hierarchy is at most $\tilde{O}(\frac{1}{\beta^2})$ per update to the current layer. This will result in an update time of $\tilde{O}((\frac{\operatorname{polylog} n}{\beta})^{2(i-1)} \cdot \frac{1}{\beta^2})$ in the i-th contracted graph per update to the input graph. The update time for maintaining the tree T will then be $\tilde{O}(\frac{1}{\beta^{2k}} + \beta^k m)$, which is $m^{2/3}$ at best, i.e., no better than the simpler approach above. A tighter analysis can improve this update time significantly: The second part of Theorem 4.1.3 bounds the amortized number

of edges to become inter-cluster edges by $\tilde{O}(\frac{1}{\beta})$. This results in an update time of $\tilde{O}((\frac{\text{polylog}\, n}{\beta})^{k+1} + \beta^k m)$. By setting $k = \sqrt{\log n}$ and $\beta = \frac{1}{m^{1/(2k+1)}}$ we can roughly balance these two terms in the update time and thus arrive at an update time of $m^{1/2+o(1)}$ while the average stretch is $n^{o(1)}$. The crux of our approach is thus an "early stopping" of the Alon et al. LDD hierarchy such that it does not "exhaust" the graph. We crucially exploit that, for an unweighted input graph, the size of the contracted graph decreases geometrically, which allows us to partially compensate for the blow-up of propagated updates in the hierarchy.

We can use the following sparsification approach to further reduce the update time to $n^{1/2+o(1)}$: The main idea is to maintain a cut sparsifier with $\tilde{O}(n)$ edges and then run the algorithm on the cut sparsifier instead of the input graph to reduce the update time from $m^{1/2+o(1)}$ to $n^{1/2+o(1)}$. The dynamic algorithm of Abraham et al. [12] can maintain such a cut sparsifier with polylogarithmic update time. Using a different cut sparsifier construction, Koutis, Levin, and Peng [167] showed in the static setting that a low-stretch tree of their cut sparsifier is also a low-stretch tree of the input graph (where the average stretch only increases multiplicatively by the approximation guarantee of the cut sparsifier). However, we cannot use exactly the same approach because the cut sparsifier of Abraham et al. has edge weights, even though the input graph is unweighted. We show that the main argument in [167] still goes through if we interpret the edge weights of the sparsifier as edge multiplicities in an unweighted graph. We then show that the algorithm of Theorem 4.1.1 can also handle such graphs for updates that increment or decrement the multiplicity of some edge by 1. A fine-grained analysis of the total multiplicity of edges of the sparsifier and its expected amount of change per update to the input graph then gives the desired result.

In Section 4.4, where we present the details of our approach, we consider two slight generalizations: First, we implicitly handle the case that the input graph could become disconnected by maintaining a low-stretch *forest*. Second, we give a parameterized analysis that also allows for a trade-off between stretch and update time.

Low Diameter Decomposition. To obtain a suitable algorithm for dynamically maintaining a low-diameter decomposition, we follow the widespread paradigm of first designing a decremental – i.e., deletions-only – algorithm and then extending it to a fully dynamic one. We can show that, for any sequence of at most m edge deletions (where m is the initial number of edges in the graph), a $(\beta, O(\frac{\log n}{\beta}))$-decomposition can be maintained with expected total update time $\tilde{O}(m/\beta)$. Here, we build upon the work of Miller et al. [195] who showed that *exponential random-shift clustering* produces clusters of radius $O(\log n/\beta)$ such that each edge has a probability of at most β to go between clusters. This clustering is obtained by first having each node sample a random *shift value* from the exponential distribution and then determining the cluster center of each node as the node to which it minimizes the difference between distance and (other node's) shift value.

In the parallel algorithm of [195], the clustering is obtained by essentially com-

puting one single-source shortest path tree of maximum depth $O(\log n/\beta)$. To make this computation efficient[3], the shift values are rounded to integer values and the fractional values are only considered for tie-breaking. We observe that one can maintain this bounded-depth shortest path tree with a simple modification of the well-known Even-Shiloach algorithm that spends time $O(\deg(v))$ every time a node v increases its level (distance from the source) in the tree. By rounding to integer edge weights, similar to [195], we can make sure that the number of level increases to consider is at most $O(\log n/\beta)$ for each node. Note however that this standard argument charging each node only when it increases its level is not enough for our purpose: the assignment of nodes to clusters follows the fractional values for tie-breaking, which might result in some node v changing its cluster – and in this way also spend time $O(\deg(v))$ – without increasing its level (note that here the difficulty is not on maintaining the cluster that v belongs to, but rather on bounding the number of cluster changes for v). As has been observed in [195], the fractional values of the shift values effectively induce a random permutation on the nodes. Using a similar argument as in the analysis of the dynamic spanner algorithm of Baswana et al. [33], we can thus show that in expectation each node changes its cluster at most $O(\log n)$ times while staying at a particular level. This results in a total update time of $\tilde{O}(m/\beta)$. Trivially, this also bounds the total number of times that edges become inter-cluster edges during the whole decremental algorithm by $\tilde{O}(m/\beta)$. Using a more sophisticated analysis we can obtain the stronger bound of $\tilde{O}(m)$ on the latter quantity: Intuitively, each endpoint of an edge changes its cluster at most $\tilde{O}(\frac{1}{\beta})$ times and after each cluster change the edge is an inter-cluster edge with probability at most β, yielding a total of $\tilde{O}(m \cdot \frac{1}{\beta} \cdot \beta)$ times that edges become inter-cluster edges. The rigorous argument is however more complicated because we cannot guarantee that the event of being an inter-cluster edge might not be independent of the event of the endpoint changing its cluster.

To obtain a fully dynamic algorithm, we observe that any LDD can tolerate a certain number of insertions to the graph. A $(\beta, O(\frac{\log n}{\beta}))$-decomposition allows at most βm inter-cluster edges and thus, if we insert $O(\beta m)$ edges to the graph without changing the decomposition, we still have an $(O(\beta), O(\frac{\log n}{\beta}))$-decomposition. We can exploit this observation by simply running a decremental algorithm, that is restarted from scratch after each phase of $\Theta(\beta m)$ updates to the graph. We then deal with edge deletions by delegating them to the decremental algorithm and we deal with edge insertions in a lazy way by doing nothing. This results in a total time of $\tilde{O}(m/\beta)$ that is amortized over $\Theta(\beta m)$ updates to the graph, i.e., amortized update time $\tilde{O}(1/\beta^2)$. Similarly, the amortized number of edges to become inter-cluster edges after an update is $\tilde{O}(1/\beta)$.

In our detailed description and analysis in Section 4.5, we first review the construction of Miller et al., and then present our decremental and fully dynamic algorithms.

[3]For their parallel algorithm, efficiency in particular means low depth of the computation tree.

Dynamic Spanner via Exponential Random Shift Clustering At a high level, the key idea behind our improved result on dynamic spanners is that a slight extension of the techniques we developed already leads to a deletions-only algorithm. Concretely, we show that it is possible to combine our decremental random-shift clustering with the recent spanner construction of Elkin and Neiman [92] to design such an algorithm. Observe that this is sufficient for our purposes due to the decomposability property of spanners, which allows to extend decremental algorithms to fully dynamic ones while paying only a logarithmic factor in the size of the spanner and the update time of the data-structure (see e.g., [33]).

Inspired by the low diameter clustering algorithm of Miller et al. [195], Elkin and Neiman devised the following simple routine for constructing a spanner: (1) each node samples a random *shift value* (which depends on some stretch parameter) from the exponential distribution and then it defines its cluster center to be the node which minimizes the difference between the distance of these two nodes and the other node's shift value, also known as the *shifted distance*; (2) for each node all the neighbours that lie on a shortest path between the node and the set of nodes whose shifted distance is within 1 of the minimum one are added to the spanner. In comparison to the low-diameter clustering, where each node needs to determine the cluster it belongs to, keeping track of the spanner edges for each node might seem more challenging at first. Fortunately, we observe that determining these edges in the static setting still reduces to computing one single-source shortest path tree of bounded depth. Moreover, similar to the random-shift clustering for low-diameter decompositions, we exploit the structural properties of this tree to maintain the spanner edges under deletions using the well-known Even-Shiloach algorithm together with the rounding tricks that were tightly linked to defining a random permutation on the nodes. Details on the implementation of this algorithm are provided in Section 4.6.

4.4 Dynamic Low Average Stretch Forest

Our dynamic algorithms for maintaining a low average stretch forest will use a hierarchy of low-diameter decompositions. We first analyze very generally the update time for maintaining such a decomposition and explain how to obtain a spanning forest from this hierarchy in a natural way, similar to the construction of Alon et al. [17]. We then analyze two different approaches for maintaining the tree, which will give us two complementary points in the design space of dynamic low-stretch tree algorithms. Finally, we explain how to exploit input graph sparsification to improve the update time of our first algorithm.

4.4.1 Generic Dynamic LDD Hierarchy

Consider some integer parameter $k \geq 1$ and parameters $\beta_0, \ldots, \beta_{k-1} \in (0, 1)$. For each $0 \leq i \leq k-1$, let $\mathcal{D}_i$ be the fully dynamic algorithm for maintaining

a $(\beta_i, O(\frac{\log n}{\beta_i}))$-decomposition as given by Theorem 4.1.3. Our *LDD-hierarchy* consists of $k+1$ multigraphs $G_0 = (V, E_0), \ldots, G_k = (V, E_k)$ where G_0 is the input graph G and, for each $0 \leq i \leq k-1$, the graph G_{i+1} is obtained from contracting G_i according to a $(\beta_i, O(\frac{\log n}{\beta_i}))$-decomposition of G_i as follows: For every node $v \in V$, let $c_i(v)$ denote the center of the cluster to which v is assigned in the $(\beta_i, O(\frac{\log n}{\beta_i}))$-decomposition of G_i. Now define E_{i+1} as the multiset of edges containing for every edge $(u, v) \in E_i$ such that $c_i(u) \neq c_i(v)$ one edge $(c_i(u), c_i(v))$, i.e., $E_{i+1} = \{(c_i(u), c_i(v)) : (u, v) \in E_i \text{ and } c_i(u) \neq c_i(v)\}$, where the multiplicity of each edge is equal to the number of edges between the corresponding clusters in G_i. Remember that all graphs G_i have the same set of nodes, but nodes that do not serve as cluster centers in G_{i-1} will be isolated in G_i. It might seem counter-intuitive at first that these isolated nodes are not removed from the graph, but observe that in our dynamic algorithm nodes might start or stop being cluster centers over time. By keeping all nodes in all subgraphs, we avoid having to explicitly deal with insertions or deletions of nodes.[4]

Note that the $(\beta_i, O(\frac{\log n}{\beta_i}))$-decomposition of G_i guarantees that $|E_{i+1}| \leq \beta_i \cdot |E_i|$ in expectation, which implies the following bound.

Observation 4.4.1. *For every $0 \leq i \leq k$, $|E_i| \leq m \cdot \prod_{0 \leq j \leq i-1} \beta_j$ in expectation.*[5]

We now analyze the update time for maintaining this LDD-hierarchy under insertions and deletions to the input graph G. Note that for each level $i \leq k-1$ of the hierarchy, changes made to the graph G_i might result in the dynamic algorithm $\mathcal{D}_i$ making changes to the $(\beta_i, O(\frac{\log n}{\beta_i}))$-decomposition of G_i. In particular, edges of G_i could start or stop being inter-cluster edges in the decomposition, which in turn leads to edges being added to or removed from G_{i+1}. Thus, a single update to the input graph G might result in a blow-up of induced updates to be processed by the algorithms $\mathcal{D}_1, \ldots, \mathcal{D}_{k-1}$. To limit this blow-up, we use an additional property of our LDD-decomposition given in Theorem 4.1.3, namely the non-trivial bound on the number of edges to become inter-cluster edges after each update.

Lemma 4.4.2. *The LDD-hierarchy can be maintained with an expected amortized update time of*

$$\tilde{O}\left(\sum_{0 \leq j \leq k-1} \frac{O(\log n)^{2(k-1)}}{\beta_j \prod_{0 \leq j' \leq j} \beta_{j'}}\right).$$

Proof. For every $0 \leq i \leq k-1$ and every $q \geq 1$ define the following random variables:

- $X_i(q)$: The total time spent by algorithm $\mathcal{D}_i$ for processing any sequence of q updates to G_i.

[4] Note that it is easy to explicitly maintain the sets of isolated and non-isolated nodes by observing the degrees.

[5] Note that for $i = 0$ the product $\prod_{0 \leq j \leq i-1} \beta_j$ is empty and thus equal to 1.

- $Y_i(q)$: The total number of changes performed to G_{i+1} by $\mathcal{D}_i$ while processing any sequence of q updates to G_i.
- $Z_i(q)$: The total time spent by algorithms $\mathcal{D}_i, \ldots, \mathcal{D}_{k-1}$ for processing any sequence of q updates to G_i.

Note that the expected values of $X_i(q)$ and $Y_i(q)$ are bounded by Theorem 4.1.3 (the latter holds since only changes involving inter-cluster edges are propagated as updates to the next level). We will show by induction on i that $\mathbb{E}[Z_i(q)] = \tilde{O}(q \cdot \sum_{i \le j \le k-1} \frac{O(\log n)^{2(k-i-1)}}{\beta_j \prod_{i \le j' \le j} \beta_{j'}})$, which with $i = 0$ implies the claim we want to prove.

Before showing the proof, observe that our LDD-hierarchy uses multiple instances of the dynamic low-diameter decomposition. We can order these instances in a hierarchical manner such that changes in the instance i only affect instances $i + 1$ and above (this is possible because all changes propagate one way through the hierarchy). Since the random bits among levels are independent, we can think of the random bits in the previous level being fixed in advance, and hence the updates to the instance i are fixed as well. The latter implies that each instance i in the LDD-hierarchy is running in the oblivious adversary setting, as required by Theorem 4.1.3.

We next prove the claimed bound on $\mathbb{E}[Z_i(q)]$. In the base case $i = k - 1$, we know by Theorem 4.1.3 that algorithm $\mathcal{D}_{k-1}$ maintaining the $(\beta_{k-1}, O(\frac{\log n}{\beta_{k-1}}))$-decomposition of G_{k-1} spends expected amortized time $\tilde{O}(\frac{1}{\beta_{k-1}^2})$ per update to G_{k-1}, i.e., $\mathbb{E}[Z_{k-1}(q)] = \mathbb{E}[X_{k-1}(q)] = \tilde{O}(q \cdot \frac{1}{\beta_{k-1}^2})$ for any $q \ge 1$. For the inductive step, consider some $0 \le i < k - 1$ and any $q \ge 1$. Any sequence of q updates to G_i induces at most $Y_i(q)$ updates to G_{i+1}. Each of those updates has to be processed by the algorithms $\mathcal{D}_{i+1}, \ldots, \mathcal{D}_{k-1}$. We thus have $Z_i(q) = X_i(q) + Z_{i+1}(Y_i(q))$.

To bound $\mathbb{E}[Z_i(q)]$, recall first the expectations of the involved random variables. As by Theorem 4.1.3 the algorithm $\mathcal{D}_i$ maintaining the $(\beta_i, O(\frac{\log n}{\beta_i}))$-decomposition of G_i has expected amortized update time $\tilde{O}(\frac{1}{\beta_i^2})$, it spends an expected total time of $\mathbb{E}[X_i(q)] = \tilde{O}(q \cdot \frac{1}{\beta_i^2})$ for any sequence of q updates to G_i. Furthermore, over the whole sequence of q updates, the expected number of edges to ever become inter-cluster edges in the $(\beta_i, O(\frac{\log n}{\beta_i}))$-decomposition of G_i is $O(q \cdot \frac{\log^2 n}{\beta_i})$. This induces at most $O(q \cdot \frac{\log^2 n}{\beta_i})$ updates (insertions or deletions) to the graph G_{i+1}, i.e., $\mathbb{E}[Y_i(q)] = O(q \cdot \frac{\log^2 n}{\beta_i})$. By the induction hypothesis, the expected amortized update time spent by $\mathcal{D}_{i+1}, \ldots, \mathcal{D}_{k-1}$ for any sequence of q' updates to G_{i+1} is $\mathbb{E}[Z_{i+1}(q')] = \tilde{O}(q' \cdot \sum_{i+1 \le j \le k-1} \frac{O(\log n)^{2(k-i-2)}}{\beta_j \prod_{i+1 \le j' \le j} \beta_{j'}})$.

Now by linearity of expectation we get

$$\mathbb{E}[Z_i(q)] = \mathbb{E}\left[X_i(q) + Z_{i+1}(Y_i(q))\right] = \mathbb{E}\left[X_i(q)\right] + \mathbb{E}\left[Z_{i+1}(Y_i(q))\right]$$

and by the law of total expectation we can bound $\mathbb{E}\left[Z_{i+1}(Y_i(q))\right]$ as follows:

$$\begin{aligned}
\mathbb{E}\left[Z_{i+1}(Y_i(q))\right] &= \sum_y \mathbb{E}\left[Z_{i+1}(Y_i(q)) \mid Y_i(q) = y\right] \cdot \mathbb{P}[Y_i(q) = y] \\
&= \sum_y \mathbb{E}\left[Z_{i+1}(y)\right] \cdot \mathbb{P}[Y_i(q) = y] \\
&= \sum_y \tilde{O}\left(y \cdot \sum_{i+1 \le j \le k-1} \frac{O(\log n)^{2(k-i-2)}}{\beta_j \prod_{i+1 \le j' \le j} \beta_{j'}}\right) \cdot \mathbb{P}[Y_i(q) = y] \\
&= \tilde{O}\left(\sum_{i+1 \le j \le k-1} \frac{O(\log n)^{2(k-i-2)}}{\beta_j \prod_{i+1 \le j' \le j} \beta_{j'}}\right) \cdot \sum_y y \cdot \mathbb{P}[Y_i(q) = y] \\
&= \tilde{O}\left(\sum_{i+1 \le j \le k-1} \frac{O(\log n)^{2(k-i-2)}}{\beta_j \prod_{i+1 \le j' \le j} \beta_{j'}}\right) \cdot \mathbb{E}[Y_i(q)] \\
&= \tilde{O}\left(\sum_{i+1 \le j \le k-1} \frac{O(\log n)^{2(k-i-2)}}{\beta_j \prod_{i+1 \le j' \le j} \beta_{j'}}\right) \cdot O\left(q \cdot \frac{\log^2 n}{\beta_i}\right) \\
&= \tilde{O}\left(q \cdot \sum_{i+1 \le j \le k-1} \frac{O(\log n)^{2(k-i-1)}}{\beta_j \prod_{i \le j' \le j} \beta_{j'}}\right)
\end{aligned}$$

We thus get

$$\begin{aligned}
\mathbb{E}[Z_i(q)] &= \tilde{O}(q \cdot \tfrac{1}{\beta_i^2}) + \tilde{O}\left(q \cdot \sum_{i+1 \le j \le k-1} \frac{O(\log n)^{2(k-i-1)}}{\beta_j \prod_{i \le j' \le j} \beta_{j'}}\right) \\
&= \tilde{O}\left(q \cdot \sum_{i \le j \le k-1} \frac{O(\log n)^{2(k-i-1)}}{\beta_j \prod_{i \le j' \le j} \beta_{j'}}\right)
\end{aligned}$$

as desired. □

Given any spanning forest T' of G_k, there is a natural way of defining a spanning forest T of G from the LDD-hierarchy. To this end, we first formally define the contraction of a node v of G to a cluster center v' of G_i (for $0 \le i \le k$) as follows: Every node v of G is contracted to itself in G_0, and, for every $1 \le i \le k$, a node v of G is contracted to v' in G_i if v is contracted to u' in G_{i-1} and $c_{i-1}(u') = v'$. Similarly, for every $0 \le i \le k$, an edge $e = (u, v)$ of G is contracted to an edge $e' = (u', v')$ of G_i if u is contracted to u' and v is contracted to v'. Now define T inductively as follows: We let T_0 be the forest consisting of the spanning trees of diamteter $O(\frac{\log n}{\beta_0})$ of the clusters in the $(\beta_0, O(\frac{\log n}{\beta_0}))$-decomposition of G_0. For every $1 \le i \le k$, we obtain T_i from T_{i-1} and a $(\beta_i, O(\frac{\log n}{\beta_i}))$-decomposition of G_i as follows: for every edge e' in a shortest path tree in one of the clusters, we include

in T_i *exactly one* edge e of G among the edges that are contracted to e' in G_i. Finally, T is obtained from T_k as follows: for every edge e' in the spanning forest T' of G_k, we include in T the edge e of G contracted to e' in G_k. As the clusters in each decomposition are non-overlapping, we are guaranteed that T is indeed a forest. Note that, apart from the time needed to maintain T', we can maintain T in the same asymptotic update time as the LDD-hierarchy (up to logarithmic factors).

We now partially analyze the stretch of T with respect to G.

Lemma 4.4.3. *For every $1 \leq i \leq k$, and for every pair of nodes u and v that are contracted to the same cluster center in G_i, there is a path from u to v in T of length at most $\frac{O(\log n)^i}{\prod_{0 \leq j \leq i-1} \beta_j}$.*

Proof. The proof is by induction on i. The induction base $i = 1$ is straightforward: For u and v to be contracted to the same cluster center in G_1, they must be contained in the same cluster C of the $(\beta_0, O(\frac{\log n}{\beta_0}))$-decomposition of G_0 maintained by $\mathcal{D}_0$. Remember that C has strong diameter at most $O(\frac{\log n}{\beta_0})$. Thus, in the shortest path tree of C there is a path of length at most $O(\frac{\log n}{\beta_0})$ from u to v using edges of $G_0 = G$. By the definition of T, this path is also present in T.

For the inductive step, let $2 \leq i \leq k$ and let u' and v' denote the cluster centers to which u and v are contracted in G_{i-1}, respectively. For u and v to be contracted to the same cluster center in G_i, u' and v' must be contained in the same cluster C of the $(\beta_{i-1}, O(\frac{\log n}{\beta_{i-1}}))$-decomposition of G_{i-1} maintained by $\mathcal{D}_{i-1}$. As C has strong diameter at most $O(\frac{\log n}{\beta_{i-1}})$, there is a path π from u' to v' of length at most $O(\frac{\log n}{\beta_{i-1}})$ in the shortest path tree of C. Let $x_1, \ldots, x_t$ denote the nodes on π, where $x_1 = u'$ and $x_t = v'$. By the definition of our tree T with respect to G, there must exist edges $(a_1, b_1), \ldots, (a_t, b_t)$ of G such that

- (a_ℓ, b_ℓ) is contained in T for all $1 \leq \ell \leq t$,
- u and a_1 are contracted to the same cluster center in G_{i-1},
- b_t and v are contracted to the same cluster center in G_{i-1}, and
- b_ℓ and $a_{\ell+1}$ are contracted to the same cluster center in G_{i-1} for all $1 \leq \ell \leq t-1$.

By the induction hypothesis we know that for every $1 \leq \ell \leq t-1$ there is a path of length at most $\frac{O(\log n)^{i-1}}{\prod_{0 \leq j \leq i-2} \beta_j}$ from b_ℓ to $a_{\ell+1}$ in T. Paths of the same maximum length also exist from u to a_1 and from b_t to v. It follows that there is a path from u to v in T of length at most

$$(t+1) \cdot \frac{O(\log n)^{i-1}}{\prod_{0 \leq j \leq i-2} \beta_j} + t \leq 3t \cdot \frac{O(\log n)^{i-1}}{\prod_{0 \leq j \leq i-2} \beta_j}$$
$$= O\left(\frac{\log n}{\beta_i}\right) \cdot \frac{O(\log n)^{i-1}}{\prod_{0 \leq j \leq i-2} \beta_j} = \frac{O(\log n)^i}{\prod_{0 \leq j \leq i-1} \beta_j}$$

as desired. □

To analyze the stretch of T, we will use the following terminology: we let the *level* of an edge e of G be the largest i such that edge e is contracted to some edge e' in G_i. Remember that E_i is a multiset of edges containing as many edges (u', v') as there are edges $(u, v) \in E$ with u and v being contracted to different cluster centers u' and v' in G_i, respectively. Thus, the expected number of edges at level i is at most $|E_i|$. Note that for an edge $e = (u, v)$ to be at level i, u and v must be contracted to the same cluster center in G_{i+1}. Therefore, by Lemma 4.4.3, the stretch of edges at level i in T with respect to G is at most $\frac{O(\log n)^{i+1}}{\prod_{0 \leq j \leq i} \beta_j}$. The expected contribution to the total stretch of T by edges at level $i \leq k-1$ is thus at most

$$|E_i| \cdot \frac{O(\log n)^{i+1}}{\prod_{0 \leq j \leq i} \beta_j} \leq \frac{m}{\beta_i} \cdot O(\log n)^{i+1} . \tag{4.1}$$

4.4.2 Dynamic Low-Stretch Tree Algorithms

To now obtain a fully dynamic algorithm for maintaining a low-stretch forest, it remains to plug in a concrete algorithm for maintaining T' together with suitable choices of the parameters. We analyze two choices for dynamically maintaining T'. The first is the "lazy" approach of recomputing a low-stretch forest from scratch after each update to the input graph. The second is a fully dynamic spanning forest algorithm with only trivial stretch guarantees.

Theorem 4.4.4 (Restatement of Theorem 4.1.1)**.** *Given any unweighted, undirected graph undergoing edge insertions and deletions, there is a fully dynamic algorithm for maintaining a spanning forest of expected average stretch $n^{o(1)}$ that has expected amortized update time $m^{1/2+o(1)}$. These guarantees hold against an oblivious adversary.*

Proof. We set $k = \lceil \sqrt{\log n} \rceil$ and $\beta_i = \beta = \frac{1}{m^{1/(2k+1)}}$ for all $0 \leq i \leq k-1$ and maintain an LDD-hierarchy with these parameters. Additionally, we maintain the graph G' induced by all non-isolated nodes of G_k, which can easily be done by maintaining the degrees of nodes in G_k. After each update to G, we compute a low-average stretch forest of T' of G'. Note that this recomputation is performed *after* having updated all graphs in the hierarchy; we use the state-of-the-art static algorithm for computing a spanning forest of the multigraph G' with total stretch $\tilde{O}(|E_k|)$ in time $\tilde{O}(|E_k|)$.

By Equation (4.1), the contribution to the total stretch of T by edges at level $i \leq k-1$ is at most $m \cdot \frac{O(\log n)^{i+1}}{\beta_i}$. To bound the contribution of edges at level k, consider some edge $e = (u, v)$ at level k and let u' and v' denote the cluster centers to which u' and v' are contracted in G_k, respectively. Let π denote the path from u' to v' in T'. Using similar arguments as in the proof of Lemma 4.4.3, the contracted nodes and edges of π can be expanded to a path from u to v in T of length at most $\frac{O(\log n)^k}{\prod_{0 \leq i \leq k-1} \beta_i} \cdot |\pi|$. Thus, the contribution of edges at level k is at most $\tilde{O}(|E_k|) \cdot$

$\frac{O(\log n)^k}{\prod_{0\le i\le k-1}\beta_i} = \tilde{O}(m \cdot O(\log n)^k)$ and the total stretch of T with respect to G is

$$\begin{aligned}
&\sum_{0\le i\le k-1} m \cdot \frac{O(\log n)^{i+1}}{\beta} + \tilde{O}(m \cdot O(\log n)^k) \\
&\qquad = \tilde{O}\left(m \cdot \left(\frac{1}{\beta} \cdot \sum_{0\le i\le k-1} O(\log n)^{i+1} + O(\log n)^k\right)\right) \\
&\qquad = \tilde{O}\left(m \cdot \frac{O(\log n)^k}{\beta}\right) \\
&\qquad = \tilde{O}\left(m m^{1/(2k+1)} \cdot O(\log n)^k\right) \\
&\qquad = m^{1+o(1)},
\end{aligned}$$

which gives an average stretch of $m^{o(1)} = n^{o(1)}$.

By Observation 4.4.1, G_k has at most $m\beta^k$ edges in expectation and thus G' has at most $m\beta^k$ nodes and edges in expectation. Using the bound of Lemma 4.4.2 for the update time of the LDD-hierarchy and the bound of $\tilde{O}(m\beta^k)$ for recomputing the low-stretch tree T' on G' from scratch, the expected amortized update time for maintaining T is

$$\begin{aligned}
\tilde{O}\left(\sum_{0\le j\le k-1} \frac{O(\log n)^{2k}}{\beta_j \prod_{0\le j'\le j}\beta_{j'}} + |E_k|\right) &= \tilde{O}\left(\sum_{0\le j\le k-1} \frac{O(\log n)^{2k}}{\beta^{j+2}} + m\beta^k\right) \\
&= \tilde{O}\left(\frac{O(\log n)^{2k}}{\beta^{k+1}} + m\beta^k\right) \\
&= \tilde{O}(m^{(k+1)/(2k+1)} \cdot O(\log n)^{2k}) \\
&= \tilde{O}(m^{1/2+1/(4k+2)} \cdot O(\log n)^{2k}) \\
&= m^{1/2+o(1)}.
\end{aligned}$$

□

Theorem 4.4.5. *Given any unweighted, undirected graph undergoing edge insertions and deletions, there is a fully dynamic algorithm for maintaining a spanning forest of expected average stretch $O(t + n^{1/3+o(1)})$ that has expected amortized update time $\frac{n^{1+o(1)}}{t}$ for every $1 \le t \le n$. These guarantees hold against an oblivious adversary.*

Proof. We set $k = \lceil \log\log n \rceil$, $\beta_0 = \sqrt{t/n}$ and $\beta_i = \sqrt{\beta_{i-1}}$ for all $1 \le i \le k-1$ and maintain an LDD-hierarchy with these parameters. The spanning forest T' is obtained by fully dynamically maintaining a spanning forest of G_k using any algorithm with polylogarithmic update time.

By Equation (4.1), the contribution to the total stretch of T by edges at level $i \le k-1$ is at most $\frac{m}{\beta_i} \cdot O(\log n)^{i+1}$. For every edge $e = (u, v)$ at level k with u contracted to u' and v contracted to v' in G_k, there is a path from u' to v' in T' that by undoing the contractions can be expanded to a path from u to v in T, which

trivially has length at most $n-1$. Thus, the contribution by each edge at level k is at most $n-1$. As for every $0 \leq i \leq k$ there are at most $|E_i| = m \cdot \prod_{0 \leq j \leq i-1} \beta_j$ edges at level i in expectation, we can bound the expected total stretch of T with respect to G as follows:

$$\begin{aligned}\sum_{0 \leq i \leq k-1} |E_i| \cdot \frac{O(\log n)^{i+1}}{\prod_{0 \leq j \leq i} \beta_j} + |E_k| \cdot n \\ &= \sum_{0 \leq i \leq k-1} \frac{m \cdot O(\log n)^{i+1}}{\beta_i} + m \cdot \prod_{0 \leq i \leq k-1} \beta_i \cdot n \\ &= m \cdot \left(\sum_{0 \leq i \leq k-1} \frac{O(\log n)^{i+1}}{\beta_i} + \prod_{0 \leq i \leq k-1} \beta_i \cdot n \right)\end{aligned}$$

This gives an average stretch of $\sum_{0 \leq i \leq k-1} \frac{O(\log n)^{i+1}}{\beta_i} + \prod_{0 \leq i \leq k-1} \beta_i \cdot n$. We now simplify these two terms. Exploiting that $\beta_i \geq \beta_0$ for all $1 \leq i \leq k-1$, we get

$$\begin{aligned}\sum_{0 \leq i \leq k-1} \frac{O(\log n)^{i+1}}{\beta_i} \leq \sum_{0 \leq i \leq k-1} \frac{O(\log n)^{i+1}}{\beta_0} = \frac{O(\log n)^k}{\beta_0} \\ &= \frac{O(\log n)^k}{\sqrt{t/n}} = \sqrt{\frac{n^{1+o(1)}}{t}}\,.\end{aligned}$$

Furthermore, the geometric progression of the β_i's gives

$$\begin{aligned}\prod_{0 \leq i \leq k-1} \beta_i \cdot n = \prod_{0 \leq i \leq k-1} \beta_0^{1/2^i} \cdot n = \beta_0^{\sum_{0 \leq i \leq k-1} 1/2^i} \cdot n = \beta_0^{2-1/2^{k-1}} \cdot n \\ &= \frac{t^{1-1/2^k}}{n^{1-1/2^k}} \cdot n \leq t \cdot n^{1/2^k} = O(t)\,.\end{aligned}$$

The average stretch of the forest maintained by our algorithm is thus at most $O(t + \sqrt{\frac{n^{1+o(1)}}{t}})$, which, after balancing the two terms, can be rewritten as $O(t + n^{1/3+o(1)})$.

It remains to bound the update time of the algorithm. By Lemma 4.4.2, the hierarchy can be maintained with an amortized update time of

$$\begin{aligned}\tilde{O}\left(\sum_{0 \leq j \leq k-1} \frac{O(\log n)^{2k}}{\beta_j \cdot \prod_{0 \leq j' \leq j} \beta_{j'}} \right) = \tilde{O}\left(\sum_{0 \leq j \leq k-1} \frac{O(\log n)^{2k}}{\beta_0^{1/2^j} \cdot \beta_0^{2-1/2^j}} \right) \\ &= \tilde{O}\left(\sum_{0 \leq j \leq k-1} \frac{O(\log n)^{2k}}{\beta_0^2} \right) \\ &= \frac{n \cdot O(\log n)^{2k}}{t} = \frac{n^{1+o(1)}}{t}\,.\end{aligned}$$

Since the amortized number of changes to G_k per update to G is trivially bounded by $\frac{n^{1+o(1)}}{t}$ as well and since T' can be maintained with polylogarithmic amortized time per update to G_k, we can maintain T with amortized update time $\frac{n^{1+o(1)}}{t}$. □

Note that the algorithm of Theorem 4.1.1 is superior to the algorithm of Theorem 4.4.5 as long as $t \leq \sqrt{n}$. If $t \geq \sqrt{n}$, then the algorithm of Theorem 4.4.5 provides stretch $O(t)$ and update time $\frac{n^{1+o(1)}}{t}$.

4.4.3 Input Graph Sparsification

In the following, we explain how input graph sparsification can be performed to the algorithm of Theorem 4.1.1 by running the algorithm on a cut sparsifier, similar to the approach of Koutis et al. [167] in the *static* setting.

Corollary 4.4.6 (Restatement of Corollary 4.1.2)**.** *Given any unweighted, undirected graph undergoing edge insertions and deletions, there is a fully dynamic algorithm for maintaining a spanning forest of expected average stretch $n^{o(1)}$ that has expected amortized update time $n^{1/2+o(1)}$. These guarantees hold against an oblivious adversary.*

To make the analysis rigorous, we introduce some additional notation for multigraphs.

Succinct Representation of Multigraphs. A multigraph $G = (V, E)$ consists of a set of nodes V and a multiset of edges E. We denote by $\bar{E} = \{(u, v) \in \binom{V}{2} \mid (u, v) \in E\}$ the support of the multiset E. This allows a multigraph $G = (V, E)$ to be succinctly represented as its *skeleton* $\bar{G} = (V, \bar{E}, \mu_G)$ where μ_G is a multiplicity function $\mu_G : \bar{E} \to \mathbb{Z}^+$ that assigns to each edge e its (positive integer) multiplicity $\mu_G(e)$. We denote by $m := |E|$ the number of multi-edges (considering multiplicities), and by $\bar{m} := |\bar{E}|$ the size of the support of E (disregarding multiplicities). For simplicity, we assume that m is polynomial in n. The total stretch of a spanning forest T is defined with respect to E, i.e.,

$$\text{stretch}_T(G) = \sum_{e=(u,v)\in E(G)} \text{dist}_T(u, v) = \sum_{e=(u,v)\in \bar{E}(G)} \mu_G(e) \cdot \text{dist}_T(u, v)\,. \quad (4.2)$$

Our dynamic algorithm will exploit that, given the skeleton of a multigraph, a low-stretch forest of can be computed without (significant) dependence on the multiplicities.

Lemma 4.4.7. *Given the skeleton $\bar{G}$ of a multigraph G, a spanning forest of G of total stretch $m^{1+o(1)}$ can be computed in time $\tilde{O}(\bar{m})$.*

Such a guarantee can be achieved with a *static* version of our algorithm, i.e., by combining the scheme of Alon et al. [17] with the LDD of Miller et al. [195]. Although we are not aware of any statement of such a "multiplicity-oblivious" running time in the literature, it seems plausible that the state-of-the art algorithms

(achieving a total stretch of $\tilde{O}(m)$) also have this property. Note however that a stretch of $m^{1+o(1)}$ is anyway good enough for our purpose.

Refined Analysis of Dynamic Low-Stretch Tree Algorithm. We now restate the guarantees of our fully dynamic low-stretch forest algorithm when the input is a multigraph undergoing insertions and deletions of multi-edges (i.e., each update increases or decreases the multiplicity of some edge by 1). Our fully dynamic LDD algorithm maintains a clustering such that every edge is an inter-cluster edge with probability β. This implies that at most a β-fraction of the edges are inter-cluster edges in expectation – regardless of whether we consider multiplicities. More precisely, contracting the clusters to single nodes yields a multigraph $G' = (V', E')$ with $|E'| \leq \beta|E|$ and $|\bar{E}'| \leq \beta|\bar{E}|$. Now, in particular the LDD hierarchy in the proof of Theorem 4.1.1 results in a multigraph $G' = (V', E')$ with $|E'| \leq \beta^k m$ and $|\bar{E}'| \leq \beta^k \bar{m}$ (after k levels). For such a graph, if its skeleton is given explicitly, one can compute a spanning forest of total stretch $O(|E'|^{1+o(1)})$ in time $\tilde{O}(|\bar{E}'|)$ by Lemma 4.4.7. Note that our dynamic algorithm can explicitly maintain the skeleton of G' with neglegible overheads in the update time. It follows that our algorithm maintains a spanning forest of total stretch $O(m^{1+o(1)})$ and has an update time of $\tilde{O}(\bar{m}^{1/2+o(1)})$.

Cut Sparsifiers. For the definition of cut sparsifiers, we consider cuts of the form $(U, V \setminus U)$ induced by a subset of nodes $U \subset V$. The capacity of such a cut $(U, V \setminus U)$ in a graph G is defined as the total multiplicity of edges crossing the cut, i.e.,

$$\mathrm{cap}_G(U, V \setminus U) = \sum_{\substack{e=(u,v)\in\bar{E} \\ u\in U, v\in V\setminus U}} \mu_G(e)$$

A $(1 \pm \epsilon)$-*cut sparsifier* [36] (with $0 \leq \epsilon \leq 1/2$) of a multigraph $G = (V, E)$ is a "subgraph" $H = (V, F)$ with $\bar{F} \subseteq \bar{E}$ such that for every $U \subset V$ we have

$$(1 - \epsilon)\,\mathrm{cap}_G(U, V \setminus U) \leq \mathrm{cap}_H(U, V \setminus U) \leq (1 + \epsilon)\,\mathrm{cap}_G(U, V \setminus U)\,,$$

i.e., H approximately preserves all cuts of G. Now let H be a $(1 \pm \epsilon)$-*cut sparsifier* of a multigraph $G = (V, E)$ and let $T = (V, E(T))$ be a (simple) spanning forest of H. For every edge e of the forest T, the nodes are naturally partitioned into two connected subsets upon removal of e. Let these two subsets be denoted by V_e and $V \setminus V_e$. Emek [93] and Koutis et al. [167], observed that by rearranging the sum in (4.2), one obtains the following cut-based characterization of the stretch:

$$\mathrm{stretch}_T(G) = \sum_{e\in E(T)} \mathrm{cap}_G(V_e, V \setminus V_e)\,.$$

Observe that the cut $(V_e, V \setminus V_e)$ is approximately preserved in H and thus $\mathrm{cap}_G(V_e, V \setminus V_e) \leq \frac{1}{1-\epsilon}\,\mathrm{cap}_H(V_e, V \setminus V_e) \leq (1 + 2\epsilon)\,\mathrm{cap}_H(V_e, V \setminus V_e)$. The

stretch of G with respect to T can now be bounded by

$$\begin{aligned}\operatorname{stretch}_T(G) &= \sum_{e \in E(T)} \operatorname{cap}_G(V_e, V \setminus V_e) \\ &\leq (1+2\epsilon) \sum_{e \in E(T)} \operatorname{cap}_H(V_e, V \setminus V_e) \\ &= (1+2\epsilon) \operatorname{stretch}_T(H)\,.\end{aligned}$$

Thus, computing the low-stretch forest on the sparsifier H instead of the original graph G only increases the total stretch by a constant factor if the number of multi-edges in H is proportional to the number of edges in G.

Dynamic Cut Sparsifiers. The fully dynamic algorithm of Abraham et al. [12] maintains, with high probability, a $(1 \pm \epsilon)$-cut sparsifier $H = (V, F)$ of a simple graph $G = (V, E)$ such that $|\bar{F}| = \tilde{O}(n/\epsilon^2)$ with update time $\operatorname{poly}(\log n, \epsilon)$. For each node v, the degree in H exceeds the degree in G by at most a factor of $(1 \pm \epsilon)$ because the cut $(\{v\}, V \setminus \{v\})$ is approximately preserved in H. We can thus bound the number of multi-edges in H (i.e., the sum of all edge multiplicities) by $|F| = O((1+\epsilon)|E|)$. The algorithm maintains a hierarchy of the edges with $O(\log n)$ layers, where edges at level i have multiplicity 4^i and each edge is at level i with probability at most $1/4^i$. After an update to the input graph, the dynamic algorithm adds or removes at most $\operatorname{poly}(\log n, \epsilon)$ edges in each level. Thus, we can bound the amount of change to H per update to G as follows: for every update to G, the expected sum of the changes to the edge multiplicities of H is at most $\operatorname{poly}(\log n, \epsilon)$.

Putting Everything Together (Proof of Corollary 4.4.6). We now first use the fully dynamic algorithm of Abraham et al. to maintain a cut sparsifier $H = (V, F)$ of the input graph $G = (V, E)$ (with $\epsilon = 1/2$) and second run our fully dynamic low-stretch tree algorithm on top of H. Here, G is a simple graph with $m = |E|$ edges and H is a multigraph with $|F| = O(m)$ and $|\bar{F}| = \tilde{O}(n)$. The spanning forest T maintained in this way gives expected total stretch at most $|F|^{1+o(1)}$ with respect to H. As argued above, this implies an expected total stretch of at most $O((1+2\epsilon)|F|^{1+o(1)}) = O(m^{1+o(1)})$ with respect to G, i.e., an average stretch of $m^{o(1)} = n^{o(1)}$. Each update to the input graph results in $\operatorname{polylog} n$ changes to the sparsifier in expectation, which are then processed as "induced" updates by our dynamic low-stretch tree algorithm. Thus, we overall arrive at an expected amortized update time of $\tilde{O}(|\bar{F}|^{1/2+o(1)}) = O(n^{1/2+o(1)})$.

4.5 Dynamic Low-Diameter Decomposition

In this section we develop our dynamic algorithm for maintaining a low-diameter decomposition following three steps. First, we review the static algorithm for constructing a low-diameter decomposition using the clustering due to Miller et

al. [195]. Second, we design a decremental algorithm by extending the Even-Shiloach algorithm [99] in a suitable way. Third, we lift our decremental algorithm to a fully dynamic one by using a "lazy" approach for handling insertions.

4.5.1 Static Low-Diameter Decomposition

In the following, we review the static algorithm for constructing a low-diameter decomposition clustering due to Miller et al. [195]. Let $G = (V, E)$ be an unweighted, undirected multigraph G, and let $\beta \in (0, 1)$ be some parameter. Our goal is to assign each node u to exactly one node $c(u)$ from V. Let $C(u) \subset V$ denote the set of nodes assigned to node u, i.e., $C(u) \coloneqq \{v \in V \mid c(v) = u\}$. For each node u, we initially set $C(u) = \varnothing$ and pick independently a *shift* value δ_u from $\mathrm{Exp}(\beta)$. Next, we assign each node u to a node v, i.e., set $c(u) = v$ and add u to $C(v)$, if v is the node that minimizes the *shifted distance* $m_v(u) \coloneqq \mathrm{dist}(u, v) - \delta_v$. Finally, we output all clusters that are non-empty. The above procedure is summarized in Algorithm 4.1.

Algorithm 4.1: Partitioning Using Exponentially Shifted Shortest Paths

Input : Multigraph $G = (V, E)$, parameter $\beta \in (0, 1)$
Output : Decomposition of G

1 For each $u \in V$, set $C(u) \leftarrow \varnothing$ and pick δ_u independently from $\mathrm{Exp}(\beta)$
2 Assign each $u \in V$ to $c(u) \leftarrow \arg\min_{v \in V} \{\mathrm{dist}(u, v) - \delta_v\}$
3 For each $v \in V$, set $C(u) \leftarrow \{v \in V \mid c(v) = u\}$
4 Return the clustering $\{C(u) \mid C(u) \neq \varnothing\}$

The following theorem gives bounds on the strong diameter and the number of inter-cluster edges output by the above partitioning.

Theorem 4.5.1 ([195], Theorem 1.2)**.** *Given an undirected, unweighted multigraph graph $G = (V, E)$ and a parameter $\beta \in (0, 1)$, Algorithm 4.1 produces a $(\beta, 2d \cdot (\log n/\beta))$-decomposition such that the guarantee on the number of inter-cluster edges holds in expectation, while the diameter bound holds with probability at least $1 - 1/n^d$, for any $d \geq 1$.*

Here, the the diameter bound holds when the maximum shift value of any node is at most $d \log n/\beta$, which happens with probability $1 - 1/n^d$. We remark that in the work of Miller et al., the above guarantees are stated only for undirected, unweighted *simple* graphs. However, by Lemma 4.4 in [195], we get that each edge $e \in E$ (regardless of whether E allows parallel edges) is an inter-cluster edges with probability at most β. By linearity of expectation, it follows that the (expected) number of inter-cluster edges in the resulting decomposition is at most βm, thus showing that the algorithm naturally extends to multigraphs.

For technical reasons, it is not sufficient in the analysis of our decremental LDD algorithm to apply Theorem 4.5.1 in a black-box manner. We thus review the crucial properties of the clustering algorithm, which we will exploit for bounding the number of changes made to inter-cluster edges in the decremental algorithm. Following [195], for each edge $e = (u, v) \in E$, let w be the *mid-point* of e, i.e., the imaginary node in the "middle" of edge e that is at distance $\frac{1}{2}$ to both u and v. Lemma 4.3 in [195] states that if u and v belong to two different clusters, i.e., $c(u) \neq c(v)$, then the shifted-distances $m_{c(u)}(w)$ and $m_{c(v)}(w)$ are within 1 of the minimum shifted distance to w.

Lemma 4.5.2 ([195]). *Let $e = (u, v)$ be an edge with mid-point w such that $c(u) \neq c(v)$ in Algorithm 4.1. Then $m_{c(u)}(w)$ and $m_{c(v)}(w)$ are within 1 of the minimum shifted distance to w.*

Lemma 4.4 of [195] shows that the probability that the smallest and the second smallest shifted distances to w are within c of each other is at most $c \cdot \beta$.

Lemma 4.5.3 ([195]). *Let $e = (u, v)$ be an edge with mid-point w. Then*

$$\mathbb{P}[|m_{c(u)}(w) - m_{c(v)}(w)| \leq c] \leq c \cdot \beta.$$

Setting $c = 1$, this gives the desired bound of β for the probability of an edge being an inter-cluster edge in Theorem 4.5.1.

Implementation. Naïvely, we could implement Algorithm 4.1 by computing $c(u)$ for each node $u \in V$ in $\tilde{O}(m)$, thus leading to a $\tilde{O}(mn)$ time algorithm. In the following, using standard techniques, we show that this running time can be reduced to $\tilde{O}(m)$.

To this end, let $\delta_{\max} \coloneqq \max_{u \in V}\{\delta_u\}$. We begin with the following augmentation of the input graph G: add a new source s to G and edges (s, u) of weight $(\delta_{\max} - \delta_u) \geq 0$, for every $u \in V$. Let $\hat{G} = (V \cup \{s\}, \hat{E}, \hat{\mathbf{w}})$ denote the resulting graph. We claim that the sub-trees below the source s in the shortest path tree of $\hat{G}$ rooted at s give us the clustering output by Algorithm 4.1 for the graph G. To see this, suppose that we instead added edges of weight $-\delta_u$ to s, for every $u \in V$. Then it is easy to check that for every $u \in V$, the distance between s and u is exactly $\min_{v \in V}(\text{dist}(u, v) - \delta_v) = \min_{v \in V} m_v(u)$. Thus the node v attaining the minimum is exactly the root of the sub-tree below the source s that contains v. Now, adding $\delta_{\max}$ to all edges incident to the source increases all distances to s by $\delta_{\max}$, and thus does not affect the shortest path tree.

Now, note that we could use Dijkstra's algorithm to construct the shortest path tree of $\hat{G}$, and modify it appropriately to output the clustering. However, for reasons that will become clear in the next section, we need to modify Dijkstra's algorithm in a specific way. This modification can be viewed as mimicking a BFS computation on a graph with special *integral* edge lengths.

We start by observing that due to the random shift values, the weight of the edges incident to the source s in $\hat{G}$ are not integers. Since we only want to deal with

integral weights, we round down all the δ_u values to $\lfloor \delta_u \rfloor$ and modify the weights of these edges using the new rounded values. Let $G' = (V \cup \{s\}, \hat{E}, \mathbf{w}')$ denote the modified graph. Note that due to the rounding, we need to introduce some tie-breaking scheme in G', such that every clustering of G' matches exactly the same clustering in $\hat{G}$, and vice versa. Naturally, the fractional parts of the rounded values, i.e., $\delta_u - \lfloor \delta_u \rfloor$, define an ordering on the nodes (if they are sorted in ascending order), and this ordering can be in turn used to break ties whenever two rounded distances are equal in G'. In their PRAM implementation, Miller et al. [195] observed that this ordering can emulated by a random permutation. This is due to the fact that the shifts are generated independently, and that the exponential distribution is memoryless.

The main motivation for using random permutations in previous works was to avoid errors that might arise from the machine precision. In our work, breaking ties according to a random permutation on the nodes is one of their algorithmic ingredients that allows us to obtain an efficient dynamic variant of the clustering. Below, we give specific implementation details about how our clustering interacts with random priorities in the static setting.

Given the graph G' and a distinguished source node $s \in V'$, Dijkstra's classical algorithm maintains an upper-bound on the shortest-path distance between each node $u \in V$ and s, denoted by $\ell(u)$. Initially, it sets $\ell(u) = \infty$, for each $u \in V$ and $\ell(s) = 0$. It also marks every node unvisited. Moreover, for each node $u \in V$, the algorithm also maintains a pointer $p(u)$ (initially set to NIL), which denotes the parent of u in the current tree rooted at s. Using these pointers, we can maintain the cluster pointer $c(u)$, for each $u \in V$. This follows from the observation that in order to compute the cluster of u, it suffices to know the cluster of its parent. Formally we have the following rule.

Observation 4.5.4. *Let $p(\cdot)$ be the parent pointers. Then for each $u \in V$, we can determine the cluster pointer $c(u)$ using the following rule:*

$$c(u) = \begin{cases} u & \text{if } p(u) = s \\ c(p(u)) & \text{otherwise.} \end{cases}$$

Now, at each iteration, Dijkstra's algorithm selects an unvisited node u with the smallest $\ell(u)$, marks it as visited, and *relaxes* all its edges. In the standard relaxation, for each edge $(u, v) \in E'$ the algorithm sets $\ell(v) \leftarrow \min\{\ell(v), \ell(u) + w'(u, v)\}$ and updates $p(v)$ accordingly. Here, we present a relaxation according to the following tie-breaking scheme. Let π be a random permutation on V. For $u, v \in V$, we write $\pi(u) < \pi(v)$ if u appears before v in the permutation π. Now, when relaxing an edge $(u, v) \in E'$, we set u to be the parent of v, i.e., $p(v) = u$, and $\ell(v) = \ell(u) + w'(u, v)$, if the following holds

$$\begin{aligned} &\ell(v) > \ell(u) + w'(u, v), \text{ or} \\ &\ell(v) = \ell(u) + w'(u, v) \text{ and } \pi(c(v)) > \pi(c(u)). \end{aligned} \tag{4.3}$$

After each edge relaxation, we also update the cluster pointers using Observation 4.5.4. We continue the algorithm until every node is visited. As usual, we maintain the unvisited nodes in a heap Q, keyed by the their estimates $\ell(v)$. This procedure is summarized in Algorithm 4.2.

Algorithm 4.2: Modified Dijkstra

Input : Graph $G' = (V \cup \{s\}, E', \mathbf{w}')$
Output : Decomposition of G

```
Generate random permutation π on V
foreach u ∈ V do
    Set ℓ(u) ← ∞
    Set p(u) ← NIL
    Set c(u) ← NIL
Set ℓ(s) ← 0
Add every u ∈ V ∪ {s} into heap Q with key ℓ(u)
while heap Q is not empty do
    Take node u with minimum key ℓ(u) from heap Q and remove it from Q
    foreach neighbor v of u do
        RELAX(u, v, w', frac)
        if p(v) = s then
            Set c(v) ← v
        else
            Set c(v) ← c(p(v))
Procedure RELAX(u, v, w', frac)
    if ℓ(v) > ℓ(u) + w'(u, v) then
        Set ℓ(v) ← ℓ(u) + w'(u, v)
        Set p(v) ← u
    else if ℓ(v) = ℓ(u) + w'(u, v) and π(c(v)) > π(c(u)) then
        Set p(v) ← u
```

Correctness of Algorithm 4.2 follows by our above discussion. Moreover, the running time of the algorithm is asymptotically bounded by the running time of Dijkstra's classical algorithm and the time to generate a random permutation. It is well known that the former runs in $\tilde{O}(m)$ time and the latter can be generated in $O(n)$ time (see e.g., Knuth Shuffle [37]), thus giving us a total $\tilde{O}(m)$ time.

4.5.2 Decremental Low-Diameter Decomposition

We now show how to maintain a lower-diameter decomposition under deletion of edges. Recall that in the previous section we observed that computing a lower-diameter decomposition of a undirected, unweighted graph can be reduced to the single-source shortest path problem in some modified graph. In the same vein, we observe that maintaining a low-diameter decomposition under edge deletions

amounts to maintaining a bounded-depth single-source shortest path tree of some modified graph under edge deletions.

Even and Shiloach [99] devised a data-structure for maintaining a bounded-depth SSSP-tree under edge deletions, which we refer to as *ES-tree*. The ES-tree initially worked only for undirected, unweighted graphs. However, later works [131, 164] observed that it can be extended even to directed, weighted graphs with positive integer edges weights. The mere usage of the ES-tree as a sub-routine will not suffice for our purposes, due to the constraints that our clustering imposes. In the following we show how to augment and modify an ES-tree that maintains a valid clustering, without degrading its running time guarantee.

Let $G = (V, E)$ be an undirected, unweighted graph for which we want to maintain a decremental $(\beta, \log n/\beta)$ decomposition, for any parameter $\beta \in (0, 1)$. Further, let $G' = (V \cup \{s\}, E', \mathbf{w}')$ be the undirected graph with integral edge weights, as defined in Section 4.5.1. Let π be a random permutation on V. By discussion in Section 4.5.1, in order to maintain a low-diameter decomposition of G it suffices to maintain a clustering of G' with π used for tie-breaking.

We describe an ES-tree that efficiently maintains a clustering of G' for a given root node s and a given distance parameter Δ. Here we set $\Delta = O(\log n/\beta)$, as by Theorem 4.5.1, the maximum distance that we run our algorithm to is bounded by $O(\log n/\beta)$. Our data-structure handles arbitrary edge deletions, and maintains the following information. First, for each node $u \in V \cup \{s\}$, we maintain a label $\ell(u)$, referred to as the *level* of u. This level of u represents the shortest path between the root s and u, i.e., $\ell(u) = \text{dist}(s, u)$. Next, for each node $u \in V$, we maintain pointers $p(u)$ and $c(u)$, which represent the parent of u in the tree and the node that u is assigned to, respectively. Finally, we also maintain the set of *potential* parents $P(u)$, for each $u \in V$, which is the set of all neighbors of u that are in the same level with the parent of u, and share the same clustering with u, i.e., a neighbor v of u belongs to $P(u)$ if v minimizes $(\ell(v) + \mathbf{w}'(u, v), \pi(c(v)))$ lexicographically, and $c(v) = c(u)$. Edge deletions in G' can possibly affect the above information for several nodes. Our algorithm adjusts these information on the nodes so as to make them valid for the modified graph.

Algorithm Description and Implementation. We give an overview and describe the implementation of Algorithm 4.3. The data-structures $\ell(\cdot)$, $p(\cdot)$ and $c(\cdot)$ are initialized using Algorithm 4.2 in Section 4.5.1. Note that for each $u \in V$, $P(u)$ can be computed by simply considering all neighbors of u in turn, and adding a neighbor v to $P(u)$ if v is a potential parent. The algorithm also maintains a heap Q whose intended use is to store nodes whose levels or clustering might need to be updated. (see procedure INITIALIZE()).

In our decremental algorithm, each node tries to maintain its level $\ell(u)$, which corresponds to its current distance to the root s, together with its cluster pointer $c(u)$ in the current graph. Concretely, we maintain the following invariant for each

node $u \in V$:

$$\ell(u) = \min\{\ell(v) + \mathbf{w}'(u, v) \mid v \text{ is a neighbor of } u\} \tag{4.4}$$

where ties among neighbors are broken according to (4.3). This invariant allows to compute the cluster pointer $c(u)$ using Observation 4.5.4. Deleting an edge incident to u might lead to a change in the values of $\ell(u)$ and $c(u)$. If this occurs, all neighbors of u are notified by u about this change, since their levels and cluster points might also change. It is well-known that the standard ES-tree can efficiently deal with changes involving the levels $\ell(\cdot)$. However, in our setting, it might be the case that an edge deletion forces a node u to change its cluster while the level $\ell(u)$ still remains the same under this deletion. This is the point where our algorithm differs from the standard ES-tree, and we next show that (1) such changes can be handled efficiently, and (2) the number of cluster changes per node, within the same level, is small in expectation.

Let us consider the deletion of an edge (u, v) (see procedure DELETE()); assume without loss of generality that $\ell(v) \leq \ell(u)$. Now note that an edge deletion might lead to a cluster change only if $v \in P(u)$. If this is the case, the algorithm first removes v from the set $P(u)$. If $P(u)$ is still non-empty, the clustering remains unaffected. However, if $P(u)$ is empty, the clustering of u will change, and the algorithm inserts u into the heap Q with key $\ell(u)$. Observe that a change in clustering of u might potentially lead to cluster changes for children of u, given that u was their only potential parent. In this way, we observe that deleting (u, v) might force changes in the clustering for many descendants of v. The algorithm handles such changes using procedure UPDATELEVELS(), which we describe below.

Procedure UPDATELEVELS() considers the nodes in Q in the order of their current level. At each iteration, it takes the node y with the smallest level $\ell(y)$ from Q. The node y computes the set of potential parents $P(y)$, by examining each neighbor of y in turn, and then adding to $P(y)$ all neighbors z that minimize $(\ell(z) + \mathbf{w}'(y, z), \pi(c(z)))$ lexicographically. Next, y sets $p(y)$ as one of nodes in $P(y)$, and updates its level by setting $\ell(y) = \ell(p(y)) + \mathbf{w}'(y, p(y))$. Having computed its parent pointer, y updates the cluster pointer using Observation 4.5.4. Specifically, if the parent of y is the source node v, then y form a new cluster itself, i.e., $c(y) = y$. Otherwise, y shares the same cluster with its parent and sets $c(y) = c(p(y))$. Finally, the algorithm determines whether the change in the clustering of y affected its neighbors. Concretely, for each neighbor x of y, it checks whether $y \in P(x)$. If this is not the case, then there is no change in the clustering of x. Otherwise, y is removed from $P(x)$, and if $P(x)$ becomes empty after this removal, the algorithm inserts x into the heap Q with key $\ell(x)$, given that Q does not already contain x.

Running Time Analysis. We first concern ourselves with the number of cluster change per node in our decremental algorithm. For any node $v \in V$, we say that the clustering *changes* for v due to an edge deletion if this deletion either increases

the level $\ell(v)$ or forces a change in the cluster pointer $c(v)$. It is well-known that the ES-tree can handle a level increase for any node v in time $O(\deg(v))$. As we will see next, we can also handle a cluster change for a node in the same level in $O(\deg(v) \log n)$ time. However, we need to ensure that the number of such cluster changes for any node and any fixed level is small, for our algorithm to be efficient. Below we argue that one can have a fairly good bound on the expected number of such changes, and this is due to the special tie-breaking scheme we use when assigning nodes to clusters.

Fix any node $v \in V$, and consider v during the sequence of edge deletions. Note that since only deletions are allowed, the level $\ell(v)$ is non-decreasing. This induces a natural partitioning of the sequence of edge deletions into subsequences such that the $\ell(v)$ remains unaffected during each subsequence. Specifically, for every node $v \in V$ and every $0 \leq i \leq \Delta$, let $S(i)$ the be subsequence of edge deletions during which $\ell(v) = i$, where $\Delta \leq O(\log n/\beta)$. The following bound on the expected number of cluster changes of v during $S(i)$ follows an argument by Baswana et al. [33].

Lemma 4.5.5. *For every node $v \in V$ and every $0 \leq i \leq \Delta$, during the entire subsequence $S(i)$, the cluster $c(v)$ of v changes at most $O(\log n)$ times, in expectation.*

Proof. Let $N_{i-1}(v)$ be the neighbors of v at level $(i-1)$, grouped according to the the clusters they belong to. This grouping naturally induces a family $\mathcal{P}$ of all potential parents sets P of v at level $(i-1)$, just before the beginning of subsequence $S(i)$. Let C be the set of the corresponding clusters centers, i.e., for each $P \in \mathcal{P}$ add $c(P)$ to C, and note that v can only join those centers during $S(i)$. Since we are considering only edge deletions, observe that when v leaves a cluster centred at some node $c \in C$, it cannot join later the same cluster c during $S(i)$.

We next bound the number of cluster changes. For each $c \in C$, there must exist an edge in the subsequence $S(i)$ whose deletion increases $\text{dist}(v, c)$, and thus c is no longer a valid cluster center for v at level i. The latter is also equivalent to some P with $c(P) = c$ becoming empty after this edge deletion. Let $\langle c_1, \ldots, c_t \rangle$ be the sequence of nodes of C *ordered* according to the time when v has no edge to a node in P_j, $1 \leq j \leq t$. We want to compute the probability that v ever joins the cluster centred at c_j during $S(i)$. Note that this event is a consequence of v changing its current cluster center $c_{j'}$ due to all parents in $P(j')$ increasing their level. According to our tie-breaking scheme in (4.3), for this to happen, c_j must be the first among all potential cluster centers $\{c_j, \ldots, c_t\}$ in the random permutation π. Since π is a uniform random permutation, the probability that c_j appears first is $1/(t-j+1)$. By linearity of expectation, the expected number of centers from C whose clusters v joins during $S(i)$ is $\sum_{j=1}^{t} \frac{1}{t-j+1} = O(\log t) = O(\log n)$. This also bounds the number of cluster changes of v during $S(i)$. □

We next bound the total update time of our decremental algorithm, and also give a bound on the total number of inter-cluster edges during its execution.

Theorem 4.5.6. *There is a decremental algorithm for maintaining a $(\beta, O(\frac{\log n}{\beta}))$-decomposition with at most $O(\beta n)$ clusters (in expectation) containing non-isolated nodes under a sequence of edge deletions in expected total update time $O(m \log^3 n/\beta)$ such that, over all deletions, each edge becomes an inter-cluster edge at most $O(\log^2 n)$ times in expectation.*

Proof. In a preprocessing step, we first repeat the sampling of the shift values until the maximum shift value is $\log n/\beta$. This event happens with probability $1 - 1/n$ (compare Theorem 4.5.1) and thus, by the waiting time bound, we need to repeat the sampling only a constant number of times. Therefore, this preprocessing takes time $O(n)$, which is subsumed in our claimed bound on the total update time.

We first note that procedure INITIALIZE() can be implemented in $O(m \log n)$ time. This is because (1) the data-structures $\ell(\cdot), p(\cdot)$ and $c(\cdot)$ are initialized using Algorithm 4.2 whose running time is bounded by $O(m \log n)$, (2) for each $u \in V$, the set $P(u)$ can by computed in $O(\deg(u))$ time, which in turn gives that all such sets can be determined in $\sum_{u \in V} O(\deg(u)) = O(m)$ time.

We next analyze the total time over the sequence of all edge deletions. Consider procedure DELETE(u, v) for deletion of an edge (u, v). If edge (u, v) does not lead to a change in the clustering of one of its endpoints, then it can be processed in $O(1)$ time. Otherwise, the end-point whose clustering has changed is inserted into heap Q, which can be implemented in $O(\log n)$ time. Now, observe that the computation time spent by procedure DELETE(u, v) is bounded by the number of nodes processed by heap Q after the deletion of edge (u, v), during procedure UPDATELEVELS(). By construction, the processed nodes are precisely those whose clustering has changed due to the deletion of (u, v), and after the processing, their new clustering its computed. A node y extracted from Q is processed in $O(\deg(y) \log n)$ time, as we will shortly argue. Therefore, we conclude that over the entire sequence of edge deletions, a node y will perform $O(\deg(y) \log n)$ amount of work, each time its clustering changes. By Lemma 4.5.5, as long as the level of y is not increased, the clustering of y will change $O(\log n)$ times, in expectation. Since there are at most $\Delta = O(\log n/\beta)$ levels, the expected number of cluster changes for y is bounded by $O((\log^2 n)/\beta)$. As our analysis applies to any node $y \in V$, we conclude that the expected total update time of our decremental algorithm is

$$\sum_{y \in V} O\left((\deg(y) \log^3 n)/\beta\right) = O\left((m \log^3 n/\beta)\right) . \tag{4.5}$$

To show our claim that each node y extracted from Q is processed in time $O(\deg(y) \log n)$, we need two observations. First, recall that $P(y)$ can be computed in $O(\deg(y))$ time, and thus the data-structures $\ell(\cdot), p(\cdot)$ and $c(\cdot)$ can be then updated in $O(1)$ time. Second, in the worst-case, y affects the clustering of all its neighbors and inserts them into Q. This step can be implemented in $O(\deg(y) \log n)$ time.

We finally show that each edge becomes an inter-cluster edge at most $O(\log^2 n)$ times in expectation. Fix some arbitrary edge $e = (x, y)$ and consider the graph G

after an arbitrary number of the adversary's deletions. We first formulate a necessary condition for e being an inter-cluster edge and give a bound on the probability of the corresponding event. Let w denote the mid-point of e, i.e., the imaginary node in the "middle" of edge e that is at distance $\frac{1}{2}$ to both u and v. Let $m_{c(x)}(w)$ and $m_{c(y)}(w)$ denote the shifted distance from w to $c(x)$ and $c(y)$ in G, respectively. We would like to argue that both $m_{c(x)}(w)$ and $m_{c(y)}(w)$ are close to the minimum shifted distance of the mid-point w. However, we cannot readily apply Lemma 4.5.2 as our algorithm does not run on G; instead it runs on G', in which the edge weights are rounded to integers. However, we can apply Lemma 4.5.2 on G' and get that $\lfloor m_{c(x)}(w) \rfloor$ and $\lfloor m_{c(y)}(w) \rfloor$ are within 1 of the minimum rounded shifted distance of the mid-point w. Thus, $|\lfloor m_{c(x)}(w) \rfloor - \lfloor m_{c(y)}(w) \rfloor| \leq 1$, which implies that $|m_{c(x)}(w) - m_{c(y)}(w)| \leq 2$. This means that $|m_{c(x)}(w) - m_{c(y)}(w)| \leq 2$ is a necessary condition for $e = (x, y)$ to be an inter-cluster edge. As the adversary is oblivious to the random choices of our algorithm, we know by Lemma 4.5.3 that $\mathbb{P}[|m_{c(x)}(w) - m_{c(y)}(w)| \leq 2] \leq 2\beta$ in each of the graphs created by the adversary's sequence of deletions.

Observe that for each of the endpoints (x and y) of e the level in our decremental algorithm is non-decreasing. Let $0 \leq i \leq \Delta$, and let $S(i)$, say of length t, be the (possibly empty) subsequence of edge deletions during which $\ell(x) = i$. We show below that the expected number of times that e becomes an inter-cluster edge during deletions in $S(i)$ is $O(\beta \log n)$. It then follows that the total number times e becomes an inter-cluster edges is $O(\log^2 n)$ by linearity of expectation: sum up the number of times e becomes an inter-cluster edge in each subsequence $S(i)$ for $0 \leq i \leq \Delta$ where $\Delta \leq O(\log n/\beta)$, and repeat the argument for the other endpoint y of e as well.

For every $1 \leq j \leq t$ define the following events:

- A_j is the event that e becomes an inter-cluster edge after the j-th deletion in $S(i)$, and was not an inter-cluster edge directly before this deletion.
- B_j is the event that at least one of the endpoints of e, x or y, changes its cluster after the j-th deletion in $S(i)$.
- C_j is the event that e is an inter-cluster edge after the j-th deletion in $S(i)$.
- D_j is the event that $|m_{c(x)}(w) - m_{c(y)}(w)| \leq 2$ after the j-th deletion in $S(i)$, where w is the mid-point of e.

Note that e can only become an inter-cluster edge if at least one of its endpoints changes its cluster. Thus, the event A_j implies the event $B_j \wedge C_j$ and therefore $\mathbb{P}[A_j] \leq \mathbb{P}[B_j \wedge C_j]$. Furthermore, by Lemma 4.5.2, the event C_j implies the event D_j. We thus have $\mathbb{P}[B_j \wedge C_j] \leq \mathbb{P}[B_j \wedge D_j]$. Observe that the event D_j only depends on the random choice of the shift values δ and that, in the fixed subsequence of deletions $S(i)$, the event B_j only depends on the random choice of the permutation π. Thus, B_j and D_j are independent and therefore $\mathbb{P}[B_j \wedge D_j] = \mathbb{P}[B_j] \cdot \mathbb{P}[D_j]$. Finally, note that the expected number of indices j such that the event B_j happens is

at most the expected number of cluster changes for both endpoints of e, as bounded by Lemma 4.5.5, and thus $\sum_{1\leq i\leq t}\mathbb{P}[B_j] = O(\log n)$ for the random permutation π. It follows that the expected number of times edge e becomes an inter-cluster edge (i.e., the expected number of indices j such that event A_j happens) is

$$\begin{aligned}\sum_{1\leq i\leq t}\mathbb{P}[A_j] &\leq \sum_{1\leq i\leq t}\mathbb{P}[B_j\wedge C_j] \leq \sum_{1\leq i\leq t}\mathbb{P}[B_j\wedge D_j] = \sum_{1\leq i\leq t}\mathbb{P}[B_j]\cdot\mathbb{P}[D_j]\\ &\leq \sum_{1\leq i\leq t}\mathbb{P}[B_j]\cdot 2\beta = 2\beta\cdot\sum_{1\leq i\leq t}\mathbb{P}[B_j] = O(\beta\log n)\,,\end{aligned}$$

where the penultimate inequality follows from Lemma 4.5.3. □

Note that in this proof, to bound the number total number of inter-cluster edges, we exploited that our two sources of randomness, the random shifts δ and the random permutation π have different purposes: δ influences whether an edge e is an inter-cluster edge and π influences the number of cluster changes of the endpoints of e. We have deliberately set up the algorithm in such a way that the independence of the corresponding events can be exploited in the proof. This is the reason why we explicitly introduced a new random permutation for tie-breaking instead of using the random shifts for this purpose as well.

Remark 4.5.7. Note that Equation (4.5) implies that the total expected update time of Theorem 4.5.6 is $O(m\log^3 n/\beta)$. For the sake of exposition, we have implemented the ES-tree using a heap, which introduces a $O(\log n)$ factor in the running time. [164] (Section 2.1.1) gives a faster implementation of the ES-tree that eliminates this extra $O(\log n)$ factor. Thus, using her technique, we can also bring down our running time to $O(m\log^2 n/\beta)$. This improvement will be particularly useful when applying our dynamic low-diameter decomposition to the construction of dynamic spanners in Section 4.6.

4.5.3 Fully Dynamic Low-Diameter Decomposition

We finally show how to extend the decremental algorithm of Theorem 4.5.6 to a fully dynamic algorithm, allowing also insertions of edges.

Theorem 4.5.8 (Restatement of Theorem 4.1.3)**.** *Given any unweighted, undirected multigraph undergoing edge insertions and deletions, there is a fully dynamic algorithm for maintaining a $(\beta, O(\frac{\log n}{\beta}))$-decomposition (with clusters of strong diameter $O(\frac{\log n}{\beta})$ and at most βm inter-cluster edges in expectation) that has expected amortized update time $O(\log^2 n/\beta^2)$. A spanning tree of diameter $O(\frac{\log n}{\beta})$ for each cluster can be maintained in the same time bound. The expected amortized number of edges to become inter-cluster edges after each update is $O(\log^2 n/\beta)$. These guarantees hold against an oblivious adversary.*

Algorithm 4.3: Modified ES-tree

// The modified ES-tree is formulated for weighted undirected graphs.

// Internal data structures:

- π: random permutation on V
- δ_v: random shift of v
- $P(v)$: the set of potential parents in the tree
- $p(v)$: for every node v a pointer to its parent in the tree
- $c(v)$: for every node v a pointer to the cluster center
- Q: global heap whose intended use is to store nodes whose levels might need to be updated

```
1  Procedure INITIALIZE()
2      Initialize using Algorithm 4.2
3      Set ℓ(v), P(v), p(v), c(v) for every node v accordingly
4  Procedure DELETE(u, v)
5      if v ∈ P(u) then
6          Remove v from P(u)
7          if P(u) = ∅ then
8              Insert u into heap Q with key ℓ(u)
9              UPDATELEVELS()
10 Procedure UPDATELEVELS()
11     while heap Q is not empty do
12         Take node y with minimum key ℓ(y) from heap Q and remove it from Q
13         Compute P(y) as the set of neighbors z of y minimizing
            (ℓ(z) + w(y, z), π(c(z))) lexicographically
14         Set p(y) as one of the nodes in P(y)
15         Set ℓ(y) ← ℓ(p(y)) + w'(y, p(y))
16         if p(y) = s then
17             Set c(y) ← y
18         else
19             Set c(y) = c(p(y))
20         foreach neighbor x of y do
21             if y ∈ P(x) then
22                 Remove y from P(x)
23                 if P(x) = ∅ then
24                     Insert x into heap Q with key ℓ(x) if Q does not already
                        contain x
```

Proof. The fully dynamic algorithm proceeds in phases, starting from an empty graph. For every $i > 1$, let m_i denote the number of edges in the graph at the beginning of phase i. After $\beta m_i/3$ updates in the graph we end phase i and start phase $i + 1$. At the beginning of each phase we re-initialize the decremental algorithm

of Theorem 4.5.6 for maintaining a $(\beta/3, 3 \cdot O(\frac{\log n}{\beta}))$-decomposition.[6] Whenever an edge is deleted from the graph, we pass the edge deletion on to the decremental algorithm. Whenever an edge is inserted to the graph, we do nothing, i.e., we deal with insertions of edges in a completely *lazy* manner.

We first analyze the ratio of inter-cluster edges at any time during phase i. First observe that the number of inter-cluster edges is at most $2\beta m_i/3$ in expectation, where at most $\beta m_i/3$ edges in expectation are contributed by the $(\beta/3, 3 \cdot O(\frac{\log n}{\beta}))$-decomposition of the decremental algorithm and at most $\beta m_i/3$ edges are contributed from inserted edges. Second, the number of edges in the graph is at least $m_i - \beta m_i/3$, as m_i is the initial number of edges and at most $\beta m_i/3$ edges have been deleted. Thus, the ratio of inter-cluster edges is at most

$$\frac{2\beta m_i/3}{m_i - \beta m_i/3} = \frac{2\beta}{3-\beta} \leq \frac{2\beta}{2+\beta-\beta} = \beta\,.$$

Our fully dynamic algorithm therefore correctly maintains a $(\beta, O(\frac{\log n}{\beta}))$-decomposition.

We now analyze the amortized update time of the algorithm. Start with an empty graph and consider a sequence of q updates. Let k denote the number of the phase after the q-th update. Then q can be written as $q = \sum_{1 \leq i < k} \beta m_i/3 + t$, where t is the number of updates in phase k. For every phase i that has been started, we spend time $O(m_i \log^2 n/\beta)$ by Theorem 4.5.6 and Remark 4.5.7. We know that $t \leq \beta m_k/3$ and in particular we also have $m_k \leq \sum_{1 \leq i \leq k-1} \beta m_i/3$ as every edge that is contained in the graph at the beginning of phase k has been inserted in one of the previous phases. We can thus bound the amortized spent by the algorithm for q updates by

$$\begin{aligned}&\frac{\sum_{1 \leq i \leq k-1} O(m_i \log^2 n/\beta) + O(m_k \log^2 n/\beta)}{\sum_{1 \leq i \leq k-1} \beta m_i/3} \\ &\quad\leq \frac{\sum_{1 \leq i \leq k-1} O(m_i \log^2 n/\beta) + O(\sum_{1 \leq i \leq k-1} m_i \log^2 n)}{\sum_{1 \leq i \leq k-1} \beta m_i/3} = O\left(\frac{\log^2 n}{\beta^2}\right).\end{aligned}$$

Finally, we analyze the amortized number of edges to become inter-cluster edges per update. For every phase i that has been started, we have a total number of $O(m_i \log^2 n)$ edges that become inter-cluster edges in the decremental algorithm by Theorem 4.5.6. Additionally, at most $\beta m_i/3 = O(m_i)$ inserted edges could also become inter-cluster edges. We can thus bound the amortized number of edges to

[6] Note that for the first constant number of updates this basically amounts to recomputation from scratch at each update.

become inter-cluster per update by

$$\frac{\sum_{1\leq i\leq k-1} O(m_i \log^2 n) + O(m_k \log^2 n)}{\sum_{1\leq i\leq k-1} \beta m_i/3} \leq \frac{\sum_{1\leq i\leq k-1} O(m_i \log^2 n) + O(\sum_{1\leq i\leq k-1} \beta m_i \log^2 n)}{\sum_{1\leq i\leq k-1} \beta m_i/3} = O\left(\frac{\log^2 n}{\beta}\right).$$

□

4.6 Dynamic Spanner Algorithm

4.6.1 Static Spanner Construction

In the following we review and adapt the static algorithm for constructing sparse low-stretch spanners due to Elkin and Neiman [92]. Let $G = (V, E)$ be an unweighted, undirected graph on n nodes, and let $k \geq 1$ be an integer. For every $u \in V$, we denote by $N(u)$ the set of all nodes incident to u. Recall that $\mathrm{Exp}(\beta)$ denotes the exponential distribution with parameter β. In what follows, we set $\beta = \log(cn)/k$, where $c > 3$ denotes the success probability. A $2k-1$-*spanner* of G is a a subgraph $H = (V, E')$ such that for every $u, v \in V$, $\mathrm{dist}_H(u, v) \leq 2k - 1 \cdot \mathrm{dist}_G(u, v)$. We refer to $2k-1$ and $|E'|$ as the *stretch* and *size* of H, respectively.

We next review some useful notation. Let δ_u be the shift value of node $u \in V$. For each $x, u \in V$, recall that $m_u(x) = \mathrm{dist}_G(x, u) - \delta_u$ is the shifted distance of x with respect to u, and let $p_u(x)$ denote the neighbor of x that lies on a shortest path from x to u. Also, for every node $x \in V$, let $m(x) = \min_{u\in V}\{m_u(x)\}$ be the minimum shifted distance. Using our clustering notation from Section 4.5, it follows that $c(x) = \arg\min_{u\in V}\{m_u(x)\}$, and thus $m(x) = m_{c(x)}(x)$.

We now present an algorithm that constructs spanners using exponential random-shift clustering. Specifically, we initially set $H = (V, \emptyset)$, and for each node $u \in V$, we independently pick a shift value δ_u from $\mathrm{Exp}(\beta)$. Then, for every $x \in V$, we add to the spanner H the following set of edges

$$C(x) = \{(x, p_u(x)) \,|\, m_u(x) \leq m(x) + 1\}. \tag{4.6}$$

The following theorem give bounds on the stretch and the size of the spanner output by the above algorithm.

Theorem 4.6.1 ([92]). *For any unweighted, undirected simple graph $G = (V, E)$ on n nodes, any integer $k \geq 1$, $c \geq 3$, there is a randomized algorithm that with probability at least $1 - \frac{2}{c}$ computes a spanner H with stretch $2k-1$ and size at most $(cn)^{1+1/k}$.*

Our analysis will rely on the following useful properties of the above algorithm.

Claim 4.6.2 ([92]). *The expected size of H is at most $(cn)^{1/k} \cdot n$.*

Claim 4.6.3 ([92]). *With probability at least* $1 - 1/c$*, it holds that* $\delta_u < k$ *for all* $u \in V$*.*

Claim 4.6.4 ([92]). *Assume* $\delta_u < k$ *for all* $u \in V$*. Then for any* $x \in V$*, if* u *is the node minimizing* $m_u(x)$*, i.e.,* $u = c(x)$*, then* $dist_G(u, x) < k$*.*

As argued by Elkin and Neiman, Claim 4.6.4 implies that the stretch of the spanner is at most $2k - 1$. Thus, the reason reason why the stretch guarantee is probabilistic is Claim 4.6.3.

Implementation. In the description of the spanner construction, it is not clear how to compute in nearly-linear time the set of edges $C(x)$ in Equation (4.6), for every node $x \in V$. To address this, we give an equivalent definition of $C(x)$, which better decouples the properties that the edges belonging to this set satisfy. Specifically, we define the set of edges

$$C'(x) = \{(x, y) \mid y \in N(x) \text{ and } m_{c(y)}(x) \leq m(x) + 1\}, \tag{4.7}$$

and then show that $C(x) = C'(x)$.

To this end, we will show that (a) $C(x) \subseteq C'(x)$ and (b) $C'(x) \subseteq C(x)$. Let $(x, y) \in C(x)$, where $y = p_u(x)$. By definition of $p_u(x)$, we have that $y \in N(x)$. We next show that $m_{c(y)}(x) \leq m(x) + 1$, which in turn proves (a). Indeed,

$$m_{c(y)}(x) = m_{c(y)}(y) + 1 = m(y) + 1 \leq m_u(y) + 1 = m_u(x) \leq m(x) + 1,$$

where the last inequality follows from Equation (4.6). For showing the other containment, i.e., proving (b), let $(x, y) \in C'(x)$. Then we need to prove that there exists some $u \in V$ such that $y = p_u(x)$ and $m_u(x) \leq m(x) + 1$. This follows by simply setting $u = c(y)$ and using Equation (4.7).

Now, similarly to the static low-diameter decomposition in Section 4.5.1, we augment the input graph G by adding a new source s to G and edges (s, x) of weight $(\delta_{max} - \delta_x) \geq 0$, for every $x \in V$, where $\delta_{\max} = \max_{x \in V}\{\delta_x\}$. Recall that in the resulting graph $\hat{G} = (V \cup \{s\}, \hat{E}, \hat{\mathbf{w}})$, for every $x \in V$, the node u attaining the minimum $m(x)$ is exactly the root of the sub-tree below the source s that contains u. Thus, we could use Dijkstra's algorithm to construct the shortest path tree of $\hat{G}$, and augment it appropriately to output the edge sets $C'(x)$, which in turn give us the spanner H.

However, in the dynamic setting, it is crucial for our algorithm to deal only with integral edge weights. To address this, we round down all the δ_u values to $\lfloor \delta_u \rfloor$ and modify the weights of the edges incident to the source s in $\hat{G}$. Let $G' = (V \cup \{s\}, E', \mathbf{w}')$ be the resulting graph, and let $\lfloor m_u(x) \rfloor$ denote the rounded shifted distances. Whenever two rounded distances are the same, we break ties using the permutation π on the nodes induced by the fractional values of the random shift values. Thus, the edge set $C'(x)$ is given by

$$C'(x) = \{(x, y) \mid y \in N(x),\ \lfloor m_{c(y)(x)} \rfloor \leq \lfloor m(x) \rfloor + 1 \text{ and } \pi(c(y)) < \pi(c(x))\}.$$

Finally, we observe that the definition of the above set can be further simplified by using the facts that $m_{c(y)}(x) = m_{c(y)}(y) + 1$ and $\lfloor m_{c(y)}(x) \rfloor \geq \lfloor m(x) \rfloor$, that is

$$\begin{aligned} C'(x) = \{(x,y) \mid y \in N(x),\ \lfloor m(y) \rfloor = \lfloor m(x) \rfloor - 1 \text{ or} \\ [\lfloor m(y) \rfloor = \lfloor m(x) \rfloor \text{ and } \pi(c(y)) < \pi(c(x))]\}\,. \end{aligned} \tag{4.8}$$

Interpreting the above set in terms of the shortest-path tree output by Dijkstra's algorithm, we get that for any $x \in V$, we add the edge (x, y) to the spanner H, if y is a neighbor one level above the level or x, or if x and y are at the same level, and the cluster y belongs to appears before in the permutation when compared to the cluster x belongs to. By Claim 4.6.4 the shortest-path tree has depth at most $2k$ with high probability.

Now observe that the randomized properties of this spanner construction only depend on the integer parts of the random shift values and the permutation π on the nodes induced by the order statistics of the fractional parts of the random shift values. Similar to the argument of Miller et al. [195] for low-diameter decompositions, it can be argued that due to memorylessness of the exponential distribution, one might as well use a uniformly sampled random permutation π instead to obtain a spanner with the same probabilistic properties.

4.6.2 Dynamic Spanner Algorithm

Spanners have a useful property called *decomposability*: Assume we are given a graph $G = (V, E)$ with a partition into two subgraphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$. If $H_1 = (V, F_1)$ is a spanner of G_1 and $H_2 = (V, F_2)$ is a spanner of G_2, both of stretch t, then $H = (V, F_1 \cup F_2)$ is a spanner of G. This property allows for a reduction that turns decremental algorithms into fully dynamic ones at the expense of logarithmic overhead in size and update time, as it has been observed by Baswana et al. [33].

Lemma 4.6.5 (Implicit in [33])**.** *If there is a decremental algorithm for maintaining a spanner of stretch t and expected size $s(n)$ with total update time $m \cdot u(m, n)$, then there is a fully dynamic algorithm for maintaining a spanner of stretch t and expected size $s(n) \cdot O(\log n)$ with amortized update time $u(m, n) \cdot O(\log n)$.*

In the remainder of this section, we explain how the techniques we developed in Section 4.5 allow for a decremental implementation of the spanner construction explained above.

Theorem 4.6.6. *Given any unweighted, undirected graph undergoing edge deletions, there is a decremental algorithm for maintaining a spanner of stretch $2k - 1$ and expected size $O(n^{1+1/k})$ that has expected total update time $O(km \log n)$. These guarantees hold against an oblivious adversary.*

Using the reduction of Lemma 4.6.5, these guarantees carry over to the fully dynamic setting.

Theorem 4.6.7 (Restatement of Theorem 4.1.4)**.** *Given any unweighted, undirected graph undergoing edge insertions and deletions, there is a fully dynamic algorithm for maintaining a spanner of stretch $2k-1$ and expected size $O(n^{1+1/k}\log n)$ that has expected amortized update time $O(k\log^2 n)$. These guarantees hold against an oblivious adversary.*

The decremental algorithm is obtained as follows: In a preprocessing step, the algorithm samples the random shift values for the nodes from the exponential distribution and additionally a uniformly random permutation π on the nodes. The sampling of the random shift values is repeated until $\delta_u < k$ for all $u \in V$. By Claim 4.6.3 this condition holds with probability at least $1 - 1/c$. Thus, by the waiting time bound, we need to repeat the sampling at most a constant number of times for the condition to hold. As each round of sampling takes time $O(n)$, this preprocessing step requires an additional $O(n)$ in the total update time.

We can then readily use Algorithm 4.3 from Section 4.5.2 to maintain a shortest path tree up to depth $2k$ from s in the graph G', as defined above. For maintaining the spanner dynamically, we need to extend the algorithm to maintain the set $C'(x)$ for every node x. Using the arguments introduced in Section 4.5.2, this can be done in a straightforward way: Every time a node x changes its level in the tree or changes its cluster $c(x)$, it (1) recomputes the set $C'(x)$ in time $O(\deg(x))$ and stores it in a hash set and (2) informs each neighbor about the change and updates the set $C'(y)$ of each neighbor y by setting the entry corresponding to the edge (x, y) accordingly. Both (1) and (2) require (expected) time $O(\deg(x))$. As the maximum level in the tree is $O(k)$ and at each node changes its clustering at a fixed level at most $O(\log n)$ times in expectation, the expected total update time of our algorithm is $O(km\log n)$ as desired.

4.7 Conclusion

In this chapter, we showed a fully dynamic algorithm that maintains a $n^{o(1)}$-stretch spanning tree in an unweighted, undirected graph with $n^{1/2+o(1)}$ amortized time per edge insertion or deletion. The core building block behind the algorithm is a dynamic algorithm that maintains a low-diameter clustering a graph. We also showed that this technique can be applied to the dynamic spanner problem, for which we improved upon the best-known update time and the size of the spanner. Our work leaves several open problems. One important problem is whether the running time can be brought down to $n^{o(1)}$. We believe that this is closely connected to how we deal with insertions in our dynamic clustering algorithm. In fact, any subroutine that outperforms our lazy insertion technique might lead to further improvements in the update time. Another interesting problem is extending our techniques to weighted, undirected graphs. A natural attempt is to extend the hierarchy due to Alon et al. [17] on weighted graphs to a dynamic setting. We remark that a black-box extension seems not to be feasible, so new ideas might be required to achieve this. Finally, the question of whether there are dynamic algorithms that maintain

low-stretch trees with poly-logarithmic stretch and sub-linear update time remains a major open problem.

CHAPTER 5

Incremental Exact Min-Cut in Poly-logarithmic Amortized Update Time

We present a deterministic incremental algorithm for *exactly* maintaining the size of a minimum cut with $O(\log^3 n \log\log^2 n)$ amortized time per edge insertion and $O(1)$ query time. This result partially answers an open question posed by Thorup [248]. It also stays in sharp contrast to a polynomial conditional lower-bound for the fully-dynamic weighted minimum cut problem. Our algorithm is obtained by combining a sparsification technique of Kawarabayashi and Thorup [157] or its recent improvement by Henzinger, Rao and Wang [136], and an exact incremental algorithm of Henzinger [127].

We also study space-efficient incremental algorithms for the minimum cut problem. Concretely, we show that there exists an $O(n \log n/\epsilon^2)$ space Monte-Carlo algorithm that can process a stream of edge insertions starting from an empty graph, and with high probability, the algorithm maintains a $(1+\epsilon)$-approximation to the minimum cut. The algorithm has $O((\alpha(n)\log^3 n)/\epsilon^2)$ amortized update-time and constant query-time, where $\alpha(n)$ stands for the inverse of Ackermann function.

5.1 Introduction

Computing a minimum cut of a graph is a fundamental algorithmic graph problem. While most of the focus has been on designing static efficient algorithms for finding a minimum cut, the dynamic maintenance of a minimum cut has also attracted increasing attention over the last two decades. The motivation for studying the dynamic setting is apparent, as real-life networks such as social or road network undergo constant and rapid changes.

Given an initial graph G, the goal of a dynamic graph algorithm is to build a data-structure that maintains G and supports update and query operations. Depending on the types of update operations we allow, dynamic algorithms are classified into three main categories: (i) *fully dynamic*, if update operations consist of both edge insertions and deletions, (ii) *incremental*, if update operations consist of edge insertions only and (iii) *decremental*, if update operations consist of edge deletions only. In this chapter, we study incremental algorithms for maintaining the size of a minimum cut of an unweighted, undirected graph (denoted by $\lambda(G) = \lambda$) supporting the following operations:

- Insert(u, v): Insert the edge (u, v) to G.
- QuerySize: Return the exact (approximate) size of a minimum cut of the current G.

For any $\alpha \geq 1$, we say that an algorithm is an α-approximation of λ if QuerySize returns a positive number k such that $\lambda \leq k \leq \alpha \cdot \lambda$. Our problem is characterized by two time measures; *query time*, which denotes the time needed to answer each query and *total update time*, which denotes the time needed to process *all* edge insertions. We say that an algorithm has an $O(t(n))$ amortized update time if it takes $O(m(t(n)))$ total update time for m edge insertions starting from an empty graph.

Related Work. For over a decade, the best known static and deterministic algorithm for computing a minimum cut was due to Gabow [107] which runs in $O(m + \lambda^2 n \log n)$ time. Kawarabayashi and Thorup [157] devised an $O(m \log^{12} n)$ time algorithm which applies only to unweighted, undirected simple graphs. Recently, Henzinger et al. [136] improved the running time to $O(m \log^2 n \log\log^2 n)$. Randomized Monte Carlo algorithms in the context of static minimum cut were initiated by Karger [151]. The best known randomized algorithm is due to Karger [150] and runs in $O(m \log^3 n)$ time.

Karger [152] was the first to study the dynamic maintenance of a minimum cut in its full generality. He devised a fully dynamic, albeit randomized, algorithm for maintaining a $\sqrt{1 + 2/\epsilon}$-approximation of the minimum cut in $\tilde{O}(n^{1/2+\epsilon})$ expected amortized time per edge operation. In the incremental setting, he showed that the update time for the same approximation ratio can be further improved to $\tilde{O}(n^{\epsilon})$. Thorup and Karger [246] improved upon the above guarantees by achieving an approximation factor of $\sqrt{2 + o(1)}$ and an $\tilde{O}(1)$ expected amortized time per edge operation.

Henzinger [127] obtained the following guarantees for the incremental minimum cut; for any $\epsilon \in (0, 1]$, (i) an $O(1/\epsilon^2)$ amortized update-time for a $(2 + \epsilon)$-approximation, (ii) an $O(\log^3 n/\epsilon^2)$ expected amortized update-time for a $(1 + \epsilon)$-approximation and (iii) an $O(\lambda \log n)$ amortized update-time for the exact minimum cut.

For minimum cut up to some poly-logarithmic size, Thorup [248] gave a fully dynamic Monte-Carlo algorithm for maintaining exact minimum cut in $\tilde{O}(\sqrt{n})$ time per edge operation. He also showed how to obtain an $1 + o(1)$-approximation of an arbitrary sized minimum cut with the same time bounds. In comparison to previous results, it is worth pointing out that his work achieves *worst-case* update times.

Lacki and Sankowski [177] studied the dynamic maintenance of the exact size of the minimum cut in planar graphs with arbitrary edge weights. They obtained a fully dynamic algorithm with $\tilde{O}(n^{5/6})$ worst-case query and update time.

There has been a growing interest in proving conditional lower bounds for dynamic problems in the last few years [7, 135]. A recent result of Nanongkai and Saranurak [201] shows the following conditional lower-bound for the *exact weighted* minimum cut assuming the Online Matrix-Vector Multiplication conjecture: for any $\epsilon > 0$, there are no fully-dynamic algorithms with polynomial-time preprocessing that can simultaneously achieve $O(n^{1-\epsilon})$ update-time and $O(n^{2-\epsilon})$ query-time.

Our Results and Technical Overview. We present two new incremental algorithms concerning the maintenance of the size of a minimum cut. Both algorithms apply to undirected, unweighted simple graphs.

Our first and main result, presented in Section 5.4, shows that there is a deterministic incremental algorithm for *exactly* maintaining the size of a minimum cut with $O(\log^3 n \log\log^2 n)$ amortized time per operation and $O(1)$ query time. This result allows us to partially answer in the affirmative a question regarding efficient dynamic algorithms for exact minimum cut posed by Thorup [248]. Additionally, it also stays in sharp contrast to the polynomial conditional lower-bound for the fully-dynamic weighted minimum cut problem of Nanongkai and Saranurak [201].

We obtain our result by heavily relying on a recent sparsification technique developed in the context of static minimum cut algorithms. Specifically, for a given simple graph G, Kawarabayashi and Thorup [157] (and subsequently Henzinger et al. [136]) designed an $\tilde{O}(m)$ procedure that contracts vertex sets of G and produces a multigraph H with considerably fewer vertices and edges while preserving some family of cuts of size up to $(3/2)\lambda(G)$. Motivated by the properties of H, the crucial observation is that it is "safe" to work entirely with graph H as long as the sequence of newly inserted edges do not increase the size of the minimum cut in H by more than $(3/2)\lambda(G)$. If the latter occurs, we recompute a new multigraph H for the current graph G. Since $\lambda(G) \leq n$, the number of such re-computations is $O(\log n)$. For maintaining the minimum-cut of H, we appeal to the exact incremental algorithm due to Henzinger [127]. Our main technical contribution is to skilfully combine these two algorithms and formally argue that such combination leads to our desirable guarantees.

Motivated by the recent work on *space-efficient* dynamic algorithms [49], we also study the efficient maintenance of the size of a minimum cut using only $\tilde{O}(n)$ space. Concretely, we present a $O(n \log n/\epsilon^2)$ space Monte-Carlo algorithm that can process a stream of edge insertions starting from an empty graph, and with

high probability, it maintains an $(1+\epsilon)$-approximation to the minimum cut in $O((\alpha(n)\log^3 n)/\epsilon^2)$ amortized update-time and constant query-time.

Note that while the streaming model also allows only $\tilde{O}(n)$ space, it is less constrained than the space efficient dynamic model since streaming algorithms do not need to maintain an explicit sparsifier at every moment, but just have enough information to construct one at the end of the stream. There have been several streaming algorithms [14, 159, 175] for maintaining a cut sparsifier, and thus $(1+\epsilon)$-approximating the minimum cut. The best bounds are due to Kyng et al. [175] who compute a stronger spectral sparsifier with $O(n\log n/\epsilon^2)$ size and $O(\log^2 n)$ amortized update-time. In comparison to our result, while our update-time is slightly worse, we can achieve constant query-time, whereas their algorithms requires $\Omega(n)$ time to answer a query.

5.2 Preliminaries

Let $G=(V,E)$ be an undirected, unweighted multi-graph with no self-loops. Two vertices x and y are *k-edge connected* if there exist k edge-disjoint paths connecting x and y. A graph G is *k-edge connected* if every pair of vertices is k-edge connected. The *local edge connectivity* $\lambda(x,y,G)$ of vertices x and y is the largest k such that x and y are k-edge connected in G. The *edge connectivity* $\lambda(G)$ of G is the largest k such that G is k-edge connected.

For a subset $S\subseteq V$ in G, the *edge cut* $E_G(S,V\setminus S)$ is a set of edges that have one endpoint in S and the other in $V\setminus S$. We may omit the subscript when clear from the context. Let $\lambda(S,G)=|E_G(S,V\setminus S)|$ be the size of the edge cut. If S is a singleton, we refer to such cut as a *trivial* cut. Two vertices x and y are *separated* by $E(S,V\setminus S)$ if they belong to different connected components of the graph induced by $E\setminus E(S,V\setminus S)$. A *minimum edge cut* of x and y is a cut of minimum size among all cuts separating x and y. A *global minimum cut* $\lambda(G)$ for G (or simply λ when G is clear from the context) is the minimum edge cut over all pairs of vertices. By Menger's Theorem [193], (a) the size of the minimum edge cut separating x and y is $\lambda(x,y,G)$, and (b) the size of the global minimum cut is equal to $\lambda(G)$.

Let n, m_0 and m_1 be the number of vertices, initial edges and inserted edges, respectively. The total number of edges m is the sum of the initial and inserted edges. Moreover, let λ and δ denote the size of the global minimum cut and the minimum degree in the final graph, respectively. Note that the minimum degree is always an upper bound on the edge connectivity, i.e., $\lambda\leq\delta$ and $m=m_0+m_1=\Omega(\delta n)$.

A subset $U\subseteq V$ is *contracted* if all vertices in U are identified with some element of U and all edges between them are discarded. For $G=(V,E)$ and a collection of vertex sets, let $H=(V(H),E(H))$ denote the graph obtained by contracting such vertex sets. Such contractions are associated with a mapping $h:V\to V(H)$. For an edge subset $N\subseteq E$, let $N_h=\{(h(a),h(b)):(a,b)\in N\}\subseteq E(H)$ be its corresponding edge subset induced by h.

Throughout, we will use the term with high probability (in short, w.h.p.) to denote the event that holds with probability at least $1 - 1/n^c$, for some positive constant c.

5.3 Sparse certificates

In this section we review a useful sparsification tool, introduced by Nagamochi and Ibaraki [199]. We first give the following definition from Benczur and Karger [36], which also appeared implicitly in [199].

Definition 5.3.1. *A sparse k-connectivity certificate, or simply a k-certificate, for an unweighted graph G with n vertices is a subgraph G' of G such that*

1. *G' consists of at most $k(n-1)$ edges, and*
2. *G' contains all edges crossing cuts of size at most k.*

Given an undirected graph $G = (V, E)$, a *(maximal) spanning forest decomposition (msfd)* $\mathcal{F}$ of order k is a decomposition of G into k edge-disjoint spanning forests F_i, $1 \leq i \leq k$, such that F_i is a (maximal) spanning forest of $G \setminus (F_1 \cup F_2 \ldots \cup F_{i-1})$. Note that $G_k = (V, \bigcup_{i \leq k} F_i)$ is a k-certificate. An msfd fulfills the following property.

Lemma 5.3.2 ([200])**.** *Let $\mathcal{F} = (F_1, \ldots, F_m)$ be an msfd of order m of a graph $G = (V, E)$, and let k be an integer with $1 \leq k \leq m$. Then for any nonempty and proper subset $S \subset V$,*

$$\lambda(S, G_k) \begin{cases} \geq k, & \text{if } \lambda(S, G) \geq k \\ = \lambda(S, G) & \text{if } \lambda(S, G) \leq k - 1. \end{cases}$$

We next present a proof of the above lemma, which closely follows the work of Nagamochi and Ibaraki [200]. We start by presenting the following helpful result.

Lemma 5.3.3. *Let $\mathcal{F} = (F_1, \ldots, F_m)$ be an msfd of order m of a graph $G = (V, E)$. Then for any edge $(u, v) \in F_j$ and any $i \leq j$, it holds that $\lambda(u, v, \bigcup_{l \leq i} F_l) \geq i$.*

Proof. Fix some edge $e = (u, v) \in F_j$. We first argue that for each $i = 1, \ldots, j-1$, the forest (V, F_i) contains some (u, v)-path. Indeed, by the maximality of the forest (V, F_i), the graph $(V, F_i \cup \{e\})$ must have some cycle C that contains e. Thus, $P = C \setminus e$ is the (u, v)-path in the forest (V, F_i). It follows that $(V, \bigcup_{l \leq i} F_l)$ has i edge-disjoint paths. Next, observe that $G_j = (V, \bigcup_{l \leq j} F_l)$ has j edge-disjoint paths, namely the $j-1$ edge disjoint paths in G_{j-1} (which does not contain the edge (u, v)) and the 1-edge path consisting of the edge (u, v). Hence, $\lambda(u, v, \bigcup_{l \leq i} F_l) \geq i$, □

Proof of Lemma 5.3.2. Assume that $\lambda(S,G) \leq k-1$. Then by definition of G_k, we know that G_k preserves any cut S of size up to k. Thus $\lambda(S,G_k) = \lambda(S,G)$.

For the other case, $\lambda(S,G) \geq k$ and assume that $\lambda(S,G_k) < \lambda(S,G)$ (otherwise the lemma follows). Then there is an edge $e = (u,v) \in E_G(S, V\setminus S) \setminus E_{G_k}(S, V\setminus S)$. Since $e \notin \bigcup_{i\leq k} F_i$, this means that e belongs to some forest F_j with $j > k$. By Lemma 5.3.3, we have that $\lambda(u,v,G_k) \geq k$. Since $(S, V \setminus S)$ separates u and v in G_k, it follows that $\lambda(S,G_k) = |E_{G_k}(S, V \setminus S)| \geq \lambda(u,v,G_k) \geq k$. □

Note that by Lemma 5.3.2 we have that $\lambda(G_k) \leq \lambda(G)$ since G_k is a subgraph of G. This implies that $\lambda(G_k) \geq \min(k, \lambda(G))$.

Nagamochi and Ibaraki [199] presented an $O(m+n)$ time algorithm (which we call a *decomposition algorithm* (DA)) to construct a special msfd, which we refer to as DA-msfd.

5.4 Incremental Exact Minimum Cut

In this section we present a deterministic incremental algorithm that exactly maintains $\lambda(G)$. The algorithm has $O(\log^3 n \log\log^2 n)$ update time, $O(1)$ query time and it applies to any undirected, unweighted simple graph $G = (V,E)$. The result is obtained by carefully combining a recent static min-cut algorithm by Kawarabayashi and Thorup [157] or its recent improvement due to Henzinger et al. [136], and the incremental min-cut algorithm of Henzinger [127]. We start by describing the maintenance of non-trivial cuts, that is, cuts with at least two vertices on both sides.

Maintaining non-trivial cuts. Kawarabayashi and Thorup [157] devised a near-linear time algorithm that contracts vertex sets of a simple input graph G and produces a sparse multi-graph H preserving all non-trivial minimum cuts of G. We refer to such a graph H as a KT-Sparsifier. Recently, Henzinger et al. [136] improved the running time for constructing H and provided better bounds on the size of H. We next define a slightly generalized version of a KT-Sparsifier, and then state the bounds achieved by these two algorithms.

Definition 5.4.1 (KT-Sparsifier)**.** *Let $G = (V,E)$ be an undirected, unweighted simple graph with n vertices, m edges and min-cut λ. A multi-graph $H = (V(H), E(H))$ is a KT-Sparsifier of G if the following holds:*

- *H has $n_H = \tilde{O}(n/\lambda)$ vertices and $m_H = \tilde{O}(m/\lambda)$ edges.*
- *H preserves all non-trivial cuts of size up to $(3/2)\lambda$ in G.*
- *H is obtained by contracting vertex sets in G.*

Theorem 5.4.2 ([157])**.** *Given an undirected, unweighted simple graph $G = (V,E)$, there is an $O(m\log^{12} n)$ time algorithm to construct a KT-Sparsifier H of G such that H has $O(n\log^4 n/\lambda)$ vertices and $O(m\log^4 n/\lambda)$ edges.*

In what follows, whenever we invoke the algorithm that constructs a KT-SPARSIFIER, we mean to invoke the algorithm from the theorem below.

Theorem 5.4.3 ([136]). *Given an undirected, unweighted simple graph $G = (V, E)$, there is an $O(m \log^2 n \log\log^2 n)$ time algorithm to construct a KT-SPARSIFIER H of G such that H has $O(n \log n/\lambda)$ vertices and $O(m \log n/\lambda)$ edges.*

As far as non-trivial cuts are concerned, Theorem 5.4.3 implies that it is safe to work on H instead of G as long as the sequence of newly inserted edges satisfies $\lambda_H \leq (3/2)\lambda$. To incrementally maintain the correct λ_H, we apply Henzinger's algorithm [127] on top of H. The basic idea to verify the correctness of the solution is to compute and store all min-cuts of H. Clearly, a solution is correct as long as an edge insertion does not increase the size of all min-cuts. If all min-cuts have increased, a new solution is computed using information about the previous solution. The steps above can be performed efficiently by making use of the cactus tree representation, which we will define shortly. The crucial observation is that whenever λ_H increases (and assuming that we can efficiently check this), instead of recomputing the cactus tree from scratch, we update intermediate structures that remained from the previous cactus tree. We next show a precise implementation of these steps.

The minimum edge cuts are stored using the *cactus tree* representation introduced by Dinitz, Karzanov and Lomonosov [82] (see also [103] for a concise proof). A cactus tree of a graph $G = (V, E)$ is a weighted graph $G_c = (V_c, E_c)$ defined as follows: There is a mapping $\phi : V \to V_c$ such that:

1. Every node in V maps to exactly one node in V_c and every node in V_c corresponds to a (possibly empty) subset of V.
2. $\phi(x) = \phi(y)$ iff x and y are $(\lambda(G) + 1)$-edge connected.
3. Every min-cut in G_c corresponds to a min-cut in G, and every min-cut in G corresponds to at least one min-cut in G_c.
4. If λ is odd, every edge of E_c has weight λ and G_c is a tree. If λ is even, no two simple cycles of G_c intersect in more than one node. Furthermore, edges that belong to a cycle have weight $\lambda/2$ while those not belonging to a cycle have weight λ.

Dinitz and Westbrook [84] showed that given a cactus tree, we can use the data structures from [110, 213] to efficiently maintain the cactus tree for fixed minimum cut size λ under edge insertions. This implies that this data-structure can be used to efficiently test whether min-cut has increased its value during edge insertions. The result is summarized in the theorem below.

Theorem 5.4.4 ([84]). *Given a cactus tree, there is an algorithm that maintains the cactus tree for fixed minimum cut size λ under u edge insertions, reporting when the minimum cut size increase to $\lambda + 1$ in $O(u + n)$ total time.*

We now turn our attention to the efficient construction and update of the cactus tree representation of a given multigraph G. To construct the cactus tree we use an algorithm due to Gabow [108], which proceeds as follows. It first computes a subgraph of G, called a *complete λ-intersection* or $I(G, \lambda)$, with at most λn edges, and then uses $I(G, \lambda)$ to compute the cactus tree. In the theorem below we state the running time for the cactus tree construction dependent on the time for computing $I(G, \lambda)$.

Theorem 5.4.5 ([108]). *Let $G = (V, E)$ be an undirected, unweighted multigraph, and assume there is an algorithm that computes $I(G, \lambda)$ in $O(T(m, n))$ time. Given $I(G, \lambda)$, the cactus tree representation of G can be constructed in $O(m)$ time. Hence, the total time for constructing the cactus tree of G is bounded by $O(T(m, n) + m)$.*

Gabow [107] devised an algorithm to compute $I(G, \lambda)$ in $O(m + \lambda^2 n \log n)$ time. Moreover, his algorithm is incremental in the sense that whenever $I(G, \lambda)$ is given as an input, the new $I(G, \lambda + 1)$ can be computed more efficiently, rather than just recomputing it from scratch. The precise statement and bounds are given in the following theorem.

Theorem 5.4.6 ([107]). *Given an undirected, unweighted multigraph $G = (V, E)$, there is an algorithm that computes $I(G, \lambda)$ in $O(m + \lambda^2 n \log n)$ time. Moreover, given $I(G, \lambda)$ and a sequence of edge insertions that increase the minimum cut by 1, the new $I(G, \lambda + 1)$ can be computed in $O(m' \log n)$ time, where m' is the number of edges in the current graph.*

Note that by combining Theorems 5.4.6 and 5.4.5 we get that the cactus tree for the initial graph can be computed in $O(m_0 + \lambda^2 n \log n)$ time, and the new cactus tree for some current graph whose minimum cut has increased can be computed in $O(m' \log n)$ time.

Maintaining trivial cuts. We remark that the multigraph H from Theorem 5.4.3 preserves only non-trivial cuts of G. If $\lambda = \delta$, then we also need a way to keep track of a trivial minimum cut. We achieve this by maintaining a minimum heap $\mathcal{H}_G$ on the vertices, where each vertex is stored with its degree. When an edge insertion is performed, the values of the edge endpoints are updated accordingly in the heap. It is well known that constructing $\mathcal{H}_G$ takes $O(n)$ time. The supported operations Min($\mathcal{H}_G$) and UpdateEndpoints($\mathcal{H}_G$,e) can be implemented in $O(1)$ and $O(\log n)$ time, respectively (see [76]). This leads to Algorithm 18.

Correctness. Let G be the current graph throughout the execution of the algorithm and let H be the corresponding multigraph maintained by the algorithm. Recall that H preserves some family of cuts from G. We say that H is *useful* if and only if there exists a minimum cut from G that is contained in the union of (a) all trivial cuts of G and (b) all cuts in H. Note that we consider H to be useful even in

Algorithm 5.1: Incremental Exact Minimum Cut

1 Compute the size λ_0 of the min-cut of G and set $\lambda^* \leftarrow \lambda_0$
Build a heap $\mathcal{H}_G$ on the vertices, where each vertex stores its degree as a key
Compute a KT-SPARSIFIER H of G and a mapping $h : V \to V_H$
Compute the size λ_H of the min-cut of H, a DA-msfd $F_1, \ldots, F_m$ of order m of H, $I(H, \lambda_H)$, and a cactus-tree of $\bigcup_{i \leq \lambda_H + 1} F_i$
2 Set $N_h \leftarrow \emptyset$
// Use the data-structure from Theorem 5.4.4 to maintain the cactus tree
while *there is at least one minimum cut of size* λ_H **do**
  Receive the next operation
  if *it is a query* **then**
    return $\min\{\lambda_H, \textsc{Min}(\mathcal{H}_G)\}$
  else if *it is the insertion of an edge* (u, v) **then**
    Update the cactus tree according to the insertion of the new edge $(h(u), h(v))$
    Add the edge $(h(u), h(v))$ to N_h and update the degrees of u and v in $\mathcal{H}_G$
Set $\lambda_H \leftarrow \lambda_H + 1$
3 **if** $\min\{\lambda_H, \textsc{Min}(\mathcal{H}_G) > (3/2)\lambda^*$ **then**
  // Full Rebuild Step
  Compute $\lambda(G)$ and set $\lambda^* \leftarrow \lambda(G)$
  Compute a KT-SPARSIFIER H of the current graph G
  Update λ_H to be the min-cut of H
  Compute a DA-msfd $F_1, \ldots, F_m$ of order m of H
  and then $I(H, \lambda_H)$ and a cactus tree of $\bigcup_{i \leq \lambda_H + 1} F_i$
else if $\lambda_H \leq (3/2)\lambda^*$ **then**
  // Partial Rebuild Step
  Compute a DA-msfd $F_1, \ldots, F_m$ of order m of $\bigcup_{i \leq (3/2)\lambda^* + 1} F_i \cup N_h$ and call the resulting forests $F_1, \ldots, F_m$
  // Update the cactus tree using Theorems 5.4.5 and 5.4.6
  Let $H' \leftarrow (V(H), E')$ be a graph with $E' \leftarrow I(H, \lambda_H - 1) \cup \bigcup_{i \leq \lambda_H + 1} F_i$
  Compute $I(H', \lambda_H)$, a cactus tree of H' and set $H \leftarrow H'$
else
  // Special Step
  while $\textsc{Min}(\mathcal{H}_G) \leq (3/2)\lambda^*$ **do**
    if *the next operation is a query* **then**
      return $\textsc{Min}(\mathcal{H}_G)$
    else
      Update the degrees of the edge endpoints in $\mathcal{H}_G$
    Goto Step 3
Goto Step 2

the *Special Step* (i.e., when $\lambda_H > (3/2)\lambda^*$), where H is not updated anymore since we are certain that the smallest trivial cut is smaller than any cut in H.

To prove the correctness of the algorithm we will show that (1) it correctly maintains a trivial min-cut at any time, (2) as long as $\lambda_H \leq (3/2)\lambda^*$, the algorithm correctly maintains all cuts of size up to $(3/2)\lambda^* + 1$ of H, and (3) H is useful as long as $\min\{\textsc{Min}(\mathcal{H}_G), \lambda_H\} \leq (3/2)\lambda^*$ (Note that when this condition fails we rebuild H).

Lemma 5.4.7. *The algorithm correctly maintains a trivial min-cut in G.*

Proof. This follows directly from the min-heap property of $\mathcal{H}_G$. □

To simplify the notation, in the following we will refer to Step 1 as a *Full Rebuild Step* (namely the initial *Full Rebuild Step*). Let $G = (V, E)$ be the current graph, and let H be the multigraph obtained by invoking KT-SPARSIFIER on G, at the time of a *Full Rebuild Step*. Now, as long as $\lambda_H \leq (3/2)\lambda^*$, suppose that the graph G and its corresponding multigraph H have undergone a sequence of edge insertions that triggered k executions of *Partial Rebuild Steps* (including Step 2), for some $k \geq 0$. Note that no *Full Rebuild Step* is executed as long as $\lambda_H \leq (3/2)\lambda^*$.

Let $H^{(k)} = (V(H), E(H^{(k)}))$ be the multigraph H *after* the k-th partial rebuild and let $H^{(0)} = H$. Let $N_h^{(k)} \subseteq E(H^{(k)})$ be the set of inserted edges in H that the algorithm maintains during the execution of the **while** loop in Step 2, after the $(k-1)$-st and before the k-th partial rebuild. Define $\tilde{H}^{(k)} = (V(H), \bigcup_{i \leq (3/2)\lambda^*+1} F_i^{(k)}) \cup N_h^{(k)})$ to be the sparsified graph that the algorithm maintains, where $F_1^{(k)}, \ldots, F_m^{(k)}$ is a DA-msfd for the graph $\tilde{H}^{(k-1)}$, and let $\tilde{H}^{(0)} = H$ be the multigraph right after the last full rebuild. We next show that $\tilde{H}^{(k)}$ preserves all cuts of size up to $(3/2)\lambda^* + 1$ of $H^{(k)}$.

Lemma 5.4.8. *For $k \geq 0$, let $H^{(k)}$ and $\tilde{H}^{(k)}$ be the multigraphs defined above. Then for any nonempty and proper subset $S \subset V(H)$,*

$$\lambda(S, \tilde{H}^{(k)}) \begin{cases} \geq (3/2)\lambda^* + 1, & \text{if } \lambda(S, H^{(k)}) \geq (3/2)\lambda^* + 1 \\ = \lambda(S, H^{(k)}) & \text{if } \lambda(S, H^{(k)}) \leq (3/2)\lambda^*. \end{cases}$$

Proof. We proceed by induction on the number k of partial rebuilds. We give the inductive step; the base case ($k = 0$) follows from the fact that $\tilde{H}^{(0)} = H = H^{(0)}$.

Fix any cut $(S, V(H) \setminus S)$ in $H^{(k)}$, and note that $H^{(k)} = (V(H), E(H^{(k-1)}) \cup N_h^{(k)})$. Define $A \coloneqq E_{H^{(k)}}(S, V(H) \setminus S) \cap N_h^{(k)}$ and $B \coloneqq E_{H^{(k)}}(S, V(H) \setminus S) \cap E(H^{(k-1)})$ such that $E_{H^{(k)}}(S, V(H) \setminus S) = A \uplus B$. Letting $F' = \bigcup_{i \leq (3/2)\lambda^*+1} F_i^{(k)}$, we similarly define edge sets $\tilde{A}$ and $\tilde{B}$ partitioning the edges $E_{\tilde{H}^{(k)}}(S, V(H) \setminus S)$ that cross the cut $(S, V(H) \setminus S)$ in $\tilde{H}^{(k)}$. Note that $A = \tilde{A}$ since edges of $N_h^{(k)}$ are always included in $\tilde{H}^{(k)}$ and $\lambda(S, H^{(k)}) = |A| + |B|$, $\lambda(S, \tilde{H}^{(k)}) = |\tilde{A}| + |\tilde{B}|$. We distinguish two cases.

First, assume $\lambda(S, H^{(k)}) \leq (3/2)\lambda^*$. Then, since $H^{(k-1)} \subseteq H^{(k)}$ and by construction of $H^{(k)}$, $\lambda(S, H^{(k-1)}) = |B|$, we get that $\lambda(S, H^{(k-1)}) \leq (3/2)\lambda^*$. By induction hypothesis, it follows that $\lambda(S, \tilde{H}^{(k-1)}) = \lambda(S, H^{(k-1)}) \leq (3/2)\lambda^*$. The latter along with Lemma 5.3.2 implies that $|\tilde{B}| = \lambda(S, \tilde{H}^{(k-1)})$, and thus $\lambda(S, \tilde{H}^{(k)}) = |\tilde{A}| + |\tilde{B}| = |A| + |B| = \lambda(S, H^{(k)})$.

Second, assume $\lambda(S, H^{(k)}) \geq (3/2)\lambda^*+1$. Then either $\lambda(S, H^{(k-1)}) \leq (3/2)\lambda^*$ or $\lambda(S, H^{(k-1)}) \geq (3/2)\lambda^* + 1$. In the first case, by induction hypothesis it follows that $\lambda(S, \tilde{H}^{(k-1)}) = \lambda(S, H^{(k-1)}) \leq (3/2)\lambda^*$. This along with Lemma 5.3.2 implies that $|\tilde{B}| = \lambda(S, \tilde{H}^{(k-1)})$, and thus $\lambda(S, \tilde{H}^{(k)}) = |\tilde{A}| + |\tilde{B}| = |A| + |B| = \lambda(S, H^{(k)}) \geq (3/2)\lambda^* + 1$. In the second case, by induction hypothesis it follows that $\lambda(S, \tilde{H}^{(k-1)}) \geq (3/2)\lambda^* + 1$. The latter along with Lemma 5.3.2 imply that $|\tilde{B}| \geq (3/2)\lambda^* + 1$, and thus $\lambda(S, \tilde{H}^{(k)}) = |\tilde{A}| + |\tilde{B}| \geq (3/2)\lambda^* + 1$, which completes the proof. □

We now show that the multigraphs $H^{(k)}$ and $\tilde{H}^{(k)}$ share the same set of minimum cuts.

Lemma 5.4.9. *Assume that $\lambda(H^{(k)}) \leq (3/2)\lambda^*$. Then a cut is a min-cut in $H^{(k)}$ iff it is a min cut in $\tilde{H}^{(k)}$.*

Proof. We first show that every non-min cut in $H^{(k)}$ is a non-min cut in $\tilde{H}^{(k)}$. By contrapositive, we get that a min-cut in $\tilde{H}^{(k)}$ is a min-cut in $H^{(k)}$.

To this end, let $(S, V(H) \setminus S)$ be a cut with $\lambda(S, H^{(k)}) \geq \lambda(H^{(k)}) + 1$ in $H^{(k)}$. Note that by assumption $\lambda(H^{(k)}) \leq (3/2)\lambda^*$. By Lemma 5.4.8 we distinguish two cases. (1) If $\lambda(S, H^{(k)}) \leq (3/2)\lambda^*$, then $\lambda(S, \tilde{H}^{(k)}) = \lambda(S, H^{(k)}) \geq \lambda(H^{(k)}) + 1$. (2) If $\lambda(S, H^{(k-1)}) \geq (3/2)\lambda^* + 1$, then $\lambda(S, \tilde{H}^{(k)}) \geq (3/2)\lambda^* + 1 \geq \lambda(H^{(k)}) + 1$. The above cases along with $\lambda(H^k) \geq \lambda(\tilde{H}^{(k)})$ give that $\lambda(S, \tilde{H}^{(k)}) \geq \lambda(\tilde{H}^{(k)}) + 1$, which in turn implies that $(S, V(H) \setminus S)$ cannot be a min-cut in $\tilde{H}^{(k)}$.

For the other direction, consider a min-cut $(D, V(H) \setminus D)$ of size $\lambda(D, \tilde{H}^{(k)})$ in $\tilde{H}^{(k)}$. Considering the cut in $H^{(k)}$ we know that $\lambda(D, H^{(k)}) \geq \lambda(H^{(k)})$. Then, similarly as above, Lemma 5.4.8 implies that $\lambda(D, \tilde{H}^{(k)}) \geq \lambda(H^{(k)})$. Since $(D, V(H) \setminus D)$ was chosen arbitrarily, we get that $\lambda(\tilde{H}^{(k)}) \geq \lambda(H^{(k)})$ must hold. The latter along with $\lambda(\tilde{H}^{(k)}) \leq \lambda(H^{(k)})$ imply that $\lambda(\tilde{H}^{(k)}) = \lambda(H^{(k)})$.

Now, let $(S, V(H) \setminus S)$ be a min-cut in $H^{(k)}$. Since $\tilde{H}^{(k)}$ is a subgraph of $H^{(k)}$ we know that $\lambda(S, \tilde{H}^{(k)}) \leq \lambda(S, H^{(k)})$. The latter along with $\lambda(\tilde{H}^{(k)}) = \lambda(H^{(k)})$ imply that

$$\lambda(S, \tilde{H}^{(k)}) \leq \lambda(S, H^{(k)}) = \lambda(H^{(k)}) = \lambda(\tilde{H}^{(k)}),$$

or, $\lambda(S, \tilde{H}^{(k)}) \leq \lambda(\tilde{H}^{(k)})$. It follows that the inequality must hold with equality since $\lambda(\tilde{H}^{(k)})$ is the value of min-cut in $\tilde{H}^{(k)}$. Thus, $(S, V(H) \setminus S)$ is also a min-cut in $\tilde{H}^{(k)}$. □

Lemma 5.4.10. *For some current graph G, let H be the current multigraph maintained by the algorithm and assume that $\lambda_H \leq (3/2)\lambda^*$, where λ^* denotes the min-cut of G at the last Full Rebuild Step. Then the value λ_H maintained by the algorithm satisfies $\lambda_H = \lambda(H)$.*

Proof. Let $\lambda(H^{(k)})$ be the value of λ_H *after* the k-th execution of partial rebuild step, for $k \geq 0$. Since, $\lambda(H^{(k)}) = \lambda(H)$, it suffices to show that $\lambda(H^{(k)})$ is correct. We proceed by induction on the number k of partial rebuilds since the last full rebuild.

We first consider the base case $k = 0$, i.e., the time right after the last full rebuild. At the beginning of a full rebuild, the algorithm computes a KT-SPARSIFIER H of G that preserves all non-trivial min-cuts of G. The value of λ_H is updated to $\lambda(H)$, a DA-msfd $F_1, \ldots, F_m$ is computed for H, and a cactus tree is constructed for $F' = \bigcup_{i \leq \lambda_H + 1} F_i$. Lemma 5.3.2 shows that a cut is a min-cut in H iff it is a min-cut in F'. The latter implies that since the cactus tree preserves the min-cuts of F', it also preserves those of H. The fact that the cactus tree algorithm correctly tells us when to increment λ_H in Step 2, we conclude that the value of λ_H after a full rebuild is set correctly.

We next give the inductive step. By induction hypothesis assume that $\lambda(H^{(k-1)})$ is correct. By Lemma 5.4.9 we get that a cut is a min-cut in $H^{(k-1)}$ iff it is a min-cut in $\tilde{H}^{(k-1)}$. Now, let $F_1^{(k)}, \ldots, F_m^{(k)}$ be the DA-msfd computed on $\tilde{H}^{(k-1)}$ during the k-th partial rebuild, and define $\tilde{F}^{(k)} = \bigcup_{i \leq \lambda(H^{(k-1)})+1} F_i^{(k)}$. Lemma 5.3.2 shows that a cut is min-cut in $\tilde{H}^{(k-1)}$ iff it is a min-cut in $\tilde{F}^{(k)}$. The two equivalences above give that every min-cut in $H^{(k-1)}$ is a min-cut $\tilde{F}^{(k)}$, and thus the graph $H'^{(k)}$ (as defined in Algorithm 18) correctly preserves all min-cuts of $H^{(k-1)}$. Given the correctness of $\lambda(H^{(k-1)})$, the properties of the cactus trees, and the fact that the incremental cactus tree algorithm correctly tells us when to increment $\lambda(H^{(k-1)})$ in Step 2, we conclude that $\lambda(H^{(k)})$ is the correct min-cut value for the graph $H^{(k)} = (V(H), E(H^{(k-1)}) \cup N_h^{(k)})$ after the k-th partial rebuild. □

Note that when $\lambda_H > (3/2)\lambda^*$, the above lemma is not guaranteed to hold as the algorithm does not execute a *Partial Rebuild Step* in this case. However, we will show below that this is not necessary for the correctness of the algorithm. The fact that we do not need to execute a *Partial Rebuild Step* in this setting is crucial for achieving our time bound.

Lemma 5.4.11. *If* $\min\{\textsc{Min}(\mathcal{H}_G), \lambda_H\} \leq 3/2\lambda^*$*, then* H *is useful.*

Proof. Let $(S', V \setminus S')$ be any non-trivial cut in G that is not in H. Such a cut must have cardinality strictly greater than $(3/2)\lambda^*$ since otherwise it would be contained in H. We show that $(S', V \setminus S')$ cannot be a minimum cut as long as $\min\{\textsc{Min}(\mathcal{H}_G), \lambda_H\} \leq (3/2)\lambda^*$ holds. We distinguish two cases.

1. If $\lambda_H \leq (3/2)\lambda^*$, then by Lemma 5.4.10 the algorithm maintains λ_H correctly. Since H is obtained from G by contracting vertex sets, there is a cut (S, V_H, S) in H, and thus in G, of value λ_H. It follows that $(S', V \setminus S')$ cannot be a minimum cut of G since $|E(S', V \setminus S')| > (3/2)\lambda^* \geq \lambda_H = \lambda(H) \geq \lambda(G)$, where the last inequality follows from the fact that H is a contraction of G.
2. If $\textsc{Min}(\mathcal{H}_G) \leq (3/2)\lambda^*$, then by Lemma 5.4.7 there is a cut of size $\textsc{Min}(\mathcal{H}_G) = \delta$ in G. Similarly, $(S', V \setminus S')$ cannot be a minimum cut of G since $|E(S', V \setminus S')| > (3/2)\lambda^* \geq \delta \geq \lambda(G)$.

Appealing to the above cases, we conclude H is useful since a min-cut of G is either contained in H or it is a trivial cut of G. □

Lemma 5.4.12. *Let G be some current graph. Then the algorithm correctly maintains $\lambda(G)$.*

Proof. Let G be some current graph and H be the current multigraph maintained by the algorithm. We will argue that $\lambda(G) = \min\{\textsc{Min}(\mathcal{H}_G), \lambda_H\}$.

If $\min\{\textsc{Min}(\mathcal{H}_G), \lambda_H\} \leq (3/2)\lambda^*$, then by Lemma 5.4.11, H is useful i.e., there exists a minimum cut of G that is contained in the union of all trivial cuts of G and all cuts in H. Lemma 5.4.7 guarantees that the algorithm correctly maintains $\textsc{Min}(\mathcal{H}_G)$, i.e., the trivial minimum cut of G. If $\lambda_H \leq (3/2)\lambda^*$, then Lemma 5.4.10 ensures that $\lambda_H = \lambda(H)$, and thus $\min\{\textsc{Min}(\mathcal{H}_G), \lambda_H\} = \lambda(G)$. If, however, $\lambda_H > (3/2)\lambda^*$ but $\min\{\textsc{Min}(\mathcal{H}_G), \lambda_H\} \leq (3/2)\lambda^*$, then $\lambda_H > \min\{\textsc{Min}(\mathcal{H}_G), \lambda_H\}$ which implies that $\min\{\textsc{Min}(\mathcal{H}_G), \lambda_H\} = \textsc{Min}(\mathcal{H}_G) = \lambda(G)$. As we argued above, the algorithm correctly maintains $\textsc{Min}(\mathcal{H}_G)$ at any time. Thus it follows that the algorithm correctly maintains $\lambda(G)$ in this case as well.

The only case that remains to consider is $\min\{\textsc{Min}(\mathcal{H}_G), \lambda_H\} > (3/2)\lambda^*$. But whenever this happens the algorithm performs a full rebuild step. After this full rebuild $\lambda(G) = \min\{\textsc{Min}(\mathcal{H}_G), \lambda_H\}$ trivially holds. □

Running Time Analysis.

Theorem 5.4.13. *Let G be a simple graph with n nodes and m_0 edges. Then the total time for inserting m_1 edges and maintaining a minimum edge cut of G is*

$$O((m_0 + m_1)\log^3 n \log\log^2 n).$$

If we start with an empty graph, the amortized time per edge insertion is $O(\log^3 n \log\log^2 n)$. The size of a minimum cut can be answered in constant time.

Proof. We first analyse Step 1. Note that building the heap $\mathcal{H}_G$ and computing λ_0 take $O(n)$ and $O(m_0 \log^2 n \log\log^2 n)$ time, respectively. Recall that $m_0 \geq \lambda_0 n$. The total running time for constructing H, $I(H, \lambda_H)$ and the cactus tree is dominated by $O((m_0 + \lambda_0^2 \cdot (n/\lambda_0))\log^2 n \log\log^2 n) = O(m_0 \log^2 n \log\log^2 n)$. Thus, the total time for Step 1 is $(m_0 \log^2 n \log\log^2 n)$.

Let $\lambda_H^0, \ldots, \lambda_H^f$ be the values that λ_H assumes in Step 2 during the execution of the algorithm in increasing order. We define Phase i to be all steps executed after Step 1 while $\lambda_H = \lambda_H^i$, excluding *Full Rebuild Steps* and *Special Steps*. Additionally, let $\lambda_0^*, \ldots, \lambda_{O(\log n)}^*$ be the values that λ^* assumes during the algorithm. We define *Superphase j* to consist of the j-th *Full Rebuild Step* along with all steps executed until the next *Full Rebuild Step*, i.e., while $\min\{\textsc{Min}(\mathcal{H}_G), \lambda_H\} \leq (3/2)\lambda_j^*$, where λ_j^* is the value of $\lambda(G)$ at the j-th *Full Rebuild Step*. Note that a superphase consists of a sequence of phases and potentially a final *Special Step*. Moreover, the algorithm executes a phase if $\lambda_H \leq (3/2)\lambda^*$.

We say that λ_H^i *belongs* to superphase j, if the i-th phase is executed during superphase j and $\lambda_H^i \leq (3/2)\lambda_j^*$. We remark that the number of vertices in H changes only at the beginning of a superphase, and remains unchanged during its lifespan.

Let n_j denote the number of vertices in some superphase j. We bound this quantity as follows:

Proposition 5.4.14. *Let j be a superphase during the execution of the algorithm. Then, we have*

$$n_j = O((n \log n)/\lambda_H^i), \text{ for all } \lambda_H^i \text{ belonging to superphase } j.$$

Proof. From Step 3 and Theorem 5.4.3 we know that $n_j = O((n \log n)/\lambda_j^*)$. Moreover, observe that $\lambda_j^* \leq \lambda_H^i$ and a phase is executed whenever $\lambda_H^i \leq (3/2)\lambda_j^*$. Thus, for all λ_H^i's belonging to superphase j, we get the following relation

$$\lambda_j^* \leq \lambda_H^i \leq (3/2)\lambda_j^*, \tag{5.1}$$

which in turn implies that $n_j = O((n \log n)/\lambda_j^*) = O((n \log n)/\lambda_H^i)$. □

For the remaining steps, we divide the running time analysis into two parts, one part corresponding to phases, and the other to superphases.

Part 1. For some superphase j, the i-th phase consists of the i-th execution of a *Partial Rebuild Step* followed by the execution of Step 2. Let u_i be the number of edge insertions in Phase i. By Theorem 5.4.4 and the fact that heap-insertions are performed in $O(\log n)$ time, it follows that the total time for Step 2 during the i-th phase is $O(n_j + u_i \log n) = O((n + u_i) \log n)$. Since $n_j = O((n \log n)/\lambda_j^*)$, we observe that $\bigcup_{i \leq (3/2)\lambda_j^* + 1} F_i \cup N_h$ has size $O(u_{i-1} + \lambda_j^* n_j) = O((u_{i-1} + n) \log n)$. Thus, the total time for computing DA-msfd in a *Partial Rebuild Step* is $O((u_{i-1} + n) \log n)$. Using Proposition 5.4.14 note that H' has $O(\lambda_H^i n_j) = O(n \log n)$ edges and thus it takes $O(n \log^2 n)$ time to compute $I(H', \lambda_H^i)$ and the new cactus tree.

The total time spent in Phase i is $O((u_{i-1} + u_i + n) \log^2 n)$. Let λ and λ_H denote the size of the minimum cut in the final graph and its corresponding multigraph, respectively. Note that $\sum_{i=1}^{\lambda} u_i \leq m_1$, $\lambda n \leq m_0 + m_1$ and recall Eqn. (5.1). This gives that the total work over all phases is

$$\sum_{i=1}^{\lambda_H} O((u_{i-1} + u_i + n) \log^2 n)$$

$$= \sum_{i=1}^{\lambda} O((u_{i-1} + u_i + n) \log^2 n) = O((m_0 + m_1) \log^2 n).$$

Part 2. The j-th superphase consists of the j-th execution of a *Full Rebuild Step* along with a possible execution of a *Special Step*, depending on whether the condition is met. In a *Full Rebuild Step*, computing $\lambda(G)$ takes $O((m_0 + m_1)\log^2 n \log\log^2 n)$ time. The total running time for constructing H, $I(H, \lambda_j^*)$ and the cactus tree is dominated by $O((m_0 + m_1 + (\lambda_j^*)^2 \cdot (n/\lambda_j^*))\log^2 n \log\log^2 n) = O((m_0 + m_1)\log^2 n \log\log^2 n)$. The running time of a *Special Step* is $O(m_1 \log n)$.

Throughout its execution, the algorithm begins a new superphase whenever $\lambda(G) = \min\{\textsc{Min}(\mathcal{H}_G), \lambda_H\} > (3/2)\lambda^*$. This implies that $\lambda(G)$ must be at least $(3/2)\lambda^*$, where λ^* is the value of $\lambda(G)$ at the last *Full Rebuild Step*. Thus, a new superphase begins whenever $\lambda(G)$ has increased by a factor of $3/2$, i.e., only $O(\log n)$ times over all insertions. This gives that the total time over all superphases is $O((m_0 + m_1)\log^3 n \log\log^2 n)$. □ □

5.5 Incremental $(1+\epsilon)$ Minimum Cut with $\tilde{O}(n)$ space

In this section we present two $\tilde{O}(n)$ space incremental Monte-Carlo algorithms that w.h.p. maintain the size of a min-cut up to a $(1+\epsilon)$-factor. Both algorithms have $\tilde{O}(1)$ update-time and $\tilde{O}(1)$, resp. $O(1)$ query-time. The first algorithm uses $O(n\log^2 n/\epsilon^2)$ space, while the second one improves the space complexity to $O(n\log n/\epsilon^2)$.

5.5.1 An $O(n\log^2 n/\epsilon^2)$ space algorithm

Our first algorithm follows an approach that was used in several previous work [127, 246, 248]. The basic idea is to maintain the min-cut up to some size k using small space. We achieve this by maintaining a sparse $(k+1)$-certificate and incorporating it into the incremental exact min-cut algorithm due to Henzinger [127], as described in Section 5.4. Finally we apply the well-known randomized sparsification result due to Karger [151] to obtain our result.

Maintaining min-cut up to size k using $O(kn)$ space. We incrementally maintain an msfd for an unweighted graph G using $k+1$ union-find data structures $\mathcal{F}_1, \ldots, \mathcal{F}_{k+1}$ (see [76]). Each $\mathcal{F}_i$ maintains a spanning forest F_i of G. Recall that $F_1, \ldots, F_{k+1}$ are edge-disjoint. When a new edge $e = (u, v)$ is inserted into G, we define i to be the first index such that $\mathcal{F}_i.\textsc{Find}(u) \neq \mathcal{F}_i.\textsc{Find}(v)$. If we found such an i, we append the edge e to the forest F_i by setting $\mathcal{F}_i.\textsc{Union}(u, v)$ and return i. If such an i cannot be found after $k+1$ steps, we simply discard edge e and return NULL. We refer to such procedure as $(k+1)$-$\textsc{Connectivity}(e)$.

It is easy to see that the forests maintained by $(k+1)$-$\textsc{Connectivity}(e)$ for every newly inserted edge e are indeed edge-disjoint. Combining this procedure with techniques from Henzinger [127] leads to the following Algorithm 19.

Algorithm 5.2: Incremental Exact Min-Cut up to size k

1 Set $\lambda \leftarrow 0$, initialize $k+1$ union-find data structures $\mathcal{F}_1, \ldots, \mathcal{F}_{k+1}$, $k+1$ empty forests $F_1, \ldots, F_{k+1}$, $I(G, \lambda)$, and an empty cactus tree
while *there is at least one minimum cut of size* λ **do**
  Receive the next operation
  if *it is a query* **then**
    return λ
  else if *it is the insertion of an edge* e **then**
    Set $i \leftarrow (k+1)$-Connectivity(e)
    if $i \neq$ NULL **then**
      Set $F_i \leftarrow F_i \cup \{e\}$
      Update the cactus tree according to the insertion of the edge e
2 Set $\lambda = \lambda + 1$
Let $G' = (V, E')$ be a graph with $E' \leftarrow I(G, \lambda - 1) \cup \bigcup_{i \leq \lambda+1} F_i$
Compute $I(G', \lambda)$ and a cactus tree of G'
Goto Step 2

The correctness of the above algorithm is immediate from Lemmas 5.4.8 and 5.4.10. The running time and query bounds follow from Theorem 8 of [127]. For the sake of completeness, we provide here a full proof.

Corollary 5.5.1. *For $k > 0$, there is an $O(kn)$ space algorithm that processes a stream of edge insertions starting from any empty graph G and maintains an exact value of $\min\{\lambda(G), k\}$. Starting from an empty graph, the total time for inserting m edges is $O(km\alpha(n)\log n)$ and queries can be answered in constant time, where $\alpha(n)$ stands for the inverse of Ackermann function.*

Proof. We first analyse Step 1. Initializing $k+1$ union-find data structures takes $O(kn)$ time. The running time for constructing $I(G, \lambda)$ and building an empty cactus tree is also dominated by $O(kn)$. Thus, the total time for Step 1 is $O(kn)$.

Let $\lambda_0, \ldots, \lambda_f$, where $\lambda_f \leq k$, be the values that λ assumes in Step 2 during the execution of the algorithm in increasing order. We define Phase i to be all steps executed while $\lambda = \lambda_i$. For $i \geq 1$, we can view Phase i as the i-th execution of Step 3 followed by the execution of Step 2. Let u_i denote the number of edge insertion in Phase i. The total time for testing the $(k+1)$-connectivity of the endpoints of the newly inserted edges, and updating the cactus tree in Step 2 is dominated by $O(n + k\alpha(n)u_i)$. Since the graph G' in Step 3 has always at most $O(kn)$ edges, the running time to compute $I(G', \lambda)$ and the cactus tree of G' is $O(kn\log n)$. Combining the above bounds, the total time spent in Phase i is $O(k(\alpha(n)u_i + n\log n))$. Thus, the total work over all phases is $O(km\alpha(n)\log n)$.

The space complexity of the algorithm is only $O(kn)$, since we always maintain at most $k+1$ spanning forests during its execution. □

Dealing with min-cuts of arbitrary size. We observe that Corollary 5.5.1 gives polylogarithmic amortized update time only for min-cuts up to some polylogarithmic size. For dealing with min-cuts of arbitrary size, we use the well-known sampling technique due to Karger [151]. This allows us to get an $(1+\epsilon)$-approximation to the value of a min-cut with high probability.

Lemma 5.5.2 ([151]). *Let G be any graph with minimum cut λ and let $p \geq 12(\log n)/(\epsilon^2\lambda)$. Let $G(p)$ be a subgraph of G obtained by including each of edge of G to $G(p)$ with probability p independently. Then the probability that the value of any cut of $G(p)$ has value more than $(1+\epsilon)$ or less than $(1-\epsilon)$ times its expected value is $O(1/n^4)$.*

For some integer $i \geq 1$, let G_i denote a subgraph of G obtained by including each edge of G to G_i with probability $1/2^i$ independently. We now have all necessary tools to present our incremental algorithm.

Algorithm 5.3: $(1+\epsilon)$-Min-Cut with $O(n\log^2 n/\epsilon^2)$ space

1 **for** $i = 0, \ldots, \lfloor \log n \rfloor$ **do**
  let G_i be an initially empty sampled subgraph
2 **Receive the next operation**
  if *it is a query* **then**
    Find the minimum j such that $\lambda(G_j) \leq k$ and **return** $2^j\lambda(G_j)/(1-\epsilon)$
  else if *it is the insertion of an edge e* **then**
    Include edge e to each G_i with probability $1/2^i$
    Maintain the exact min cut of each G_i up to size $k \leftarrow 48\log n/\epsilon^2$
    using Algorithm 19
3 **Goto** Step 2.

Theorem 5.5.3. *There is an $O(n\log^2 n/\epsilon^2)$ space randomized algorithm that processes a stream of edge insertions starting from an empty graph G and maintains a $(1+\epsilon)$-approximation to a min-cut of G with high probability. The amortized update time per operation is $O(\alpha(n)\log^3 n/\epsilon^2)$ and queries can be answered in $O(\log n)$ time.*

Proof. We first prove the correctness of the algorithm. For an integer $t \geq 0$, let $G^{(t)} = (V, E^{(t)})$ be the graph after the first t edge insertions. Further, let $\lambda(G^{(t)})$ denote the min-cut of $G^{(t)}$, $p^{(t)} = 12(\log n)/(\epsilon^2\lambda^{(t)})$ and $\lambda(S, G) = |E_G(S, V\setminus S)|$, for some cut $(S, V \setminus S)$. For any integer $i \leq \lfloor \log_2 1/p^{(t)} \rfloor$, Lemma 5.5.2 implies that for any cut $(S, V \setminus S)$, $((1-\epsilon)/2^i)\lambda(S, G^{(t)}) \leq \lambda(S, G_i^{(t)}) \leq ((1+\epsilon)/2^i)\lambda(S, G^{(t)})$, with probability $1 - O(1/n^4)$. Let $(S^*, V \setminus S^*)$ be a min-cut of $G_i^{(t)}$, i.e., $\lambda(S^*, G_i^{(t)}) = \lambda(G_i^{(t)})$. Setting $i = \lfloor \log_2 1/p^{(t)} \rfloor$, we get that:

$$\mathbb{E}[\lambda(G_i^{(t)})] \leq \lambda(G^{(t)})/2^i \leq 2p^{(t)}\lambda(G^{(t)}) \leq 24\log n/\epsilon^2.$$

The latter along with the implication of Lemma 5.5.2 give that for any $\epsilon \in (0, 1)$, the size of the minimum cut in $G_i^{(t)}$ is at most $(1+\epsilon)24 \log n/\epsilon^2 \leq 48 \log n/\epsilon^2$ with probability $1 - O(1/n^4)$. Thus, $j \leq \lfloor \log_2 1/p^{(t)} \rfloor$ with probability $1 - O(1/n^4)$. Additionally, we observe that the algorithm returns a $(1 + O(\epsilon))$-approximation to a min-cut of $G^{(t)}$ w.h.p. since by Lemma 5.5.2, $2^i \lambda(G_i^{(t)})/(1-\epsilon) \leq (1+\epsilon)/(1-\epsilon)\lambda(G^{(t)}) = (1 + O(\epsilon))\lambda(G^{(t)})$ w.h.p. Note that for any t, $\lfloor \log_2 1/p^{(t)} \rfloor \leq \lfloor \log n \rfloor$, and thus it is sufficient to maintain only $O(\log n)$ sampled subgraphs.

Since our algorithm applies to unweighted simple graphs, we know that $t \leq O(n^2)$. Now applying union bound over all $t \in \{1, \ldots O(n^2)\}$ gives that the probability that the algorithm does not maintain a $1 + O(\epsilon)$-approximation is at most $O(1/n^2)$.

The total expected time for maintaining a sampled subgraph is $O(m\alpha(n) \log^2 n/\epsilon^2)$ and the required space is $O(n \log n/\epsilon^2)$ (Corollary 5.5.1). Maintaining $O(\log n)$ such subgraphs gives an $O(\alpha(n) \log^3 n/\epsilon^2)$ amortized time per edge insertion and an $O(n \log^2 n/\epsilon^2)$ space requirement. The $O(\log n)$ query time follows as in the worst case we scan at most $O(\log n)$ subgraphs, each answering a min-cut query in constant time. □

5.5.2 Improving the space to $O(n \log n/\epsilon^2)$

We next show how to bring down the space requirement of the previous algorithm to $O(n \log n/\epsilon^2)$ without degrading its running time. The main idea is to keep a single sampled subgraph instead of $O(\log n)$ of them.

Let $G = (V, E)$ be an unweighted undirected graph and assume each edge is given some random weight p_e chosen uniformly from $[0, 1]$. Let G^w be the resulting weighted graph. For any $p > 0$, we denote by $G(p)$ the unweighted subgraph of G that consists of all edges that have weight at most p. We state the following lemma due to Karger [149]:

Lemma 5.5.4. *Let $k = 48 \log n/\epsilon^2$. Given a connected graph G, let p be a value such that $p \geq k/(4\lambda(G))$. Then with high probability, $\lambda(G(p)) \leq k$ and $\lambda(G(p))/p$ is an $(1+\epsilon)$-approximation to a min-cut of G.*

Proof. Since the weight of every edge is uniformly distributed, the probability that an edge has weight at most p is exactly p. Thus, $G(p)$ is a graph that contains every edge of G with probability p. The claim follows from Lemma 5.5.2. □

For any graph G and some appropriate weight $p \geq k/(4\lambda(G))$, the above lemma tells us that the min-cut of $G(p)$ is bounded by k with high probability. Thus, instead of considering the graph G along with its random edge weights, we build a collection of $k + 1$ minimum edge-disjoint spanning forests (using those edge weights). We note that such a collection is an msfd of order $k + 1$ for G with $O(kn)$ edges and by Lemma 5.4.8, it preserves all minimum cuts of G up to size k.

Our algorithm uses the following two data structures:

(1) NI-SPARSIFIER(k) data-structure: Given a graph G, where each edge e is assigned some weight p_e and some parameter k, we devise an insertion-only data-structure that maintains a collection of $k+1$ minimum edge-disjoint spanning forests $F_1, \ldots, F_{k+1}$ with respect to the edge weights. Let $F = \bigcup_{i \leq k+1} F_i$. Since we are in the incremental setting, it is known that the problem of maintaining a single minimum spanning forest can be solved in time $O(\log n)$ per insertion using the dynamic tree structure of Sleator and Tarjan [232]. Specifically, we use this data-structure to determine for each pair of nodes (u, v) the maximum weight of an edge in the cycle that the edge (u, v) induces in the minimum spanning forest F_i. Let max-weight$(F_i(u, v))$ denote such a maximum weight. The update operation works as follows: when a new edge $e = (u, v)$ is inserted into G, we first use the dynamic tree data structure to test whether u and v belong to the same tree. If no, we link their two trees with the edge (u, v) and return the pair (TRUE, NULL) to indicate that e was added to F_i and no edge was evicted from F_i. Otherwise, we check whether $p_e >$ max-weight$(F_i(e))$. If the latter holds, we make no changes in the forest and return (FALSE, e). Otherwise, we replace one of the maximum edges, say e', on the path between u and v in the tree by e and return (TRUE, e'). The boolean value that is returned indicates whether e belongs to F_i or not, the second value that is returned gives an edge that does not (or no longer) belong to F_i. Note that each edge insertion requires $O(\log n)$ time. We refer to this insert operation as INSERT-MSF(F_i, e, p_e).

Now, the algorithm that maintains the weighted minimum spanning forests implements the following operations:

- INITIALIZE-NI(k): Initializes the data structure for $k+1$ empty minimum spanning forests.
- INSERT-NI(e, p_e): Set $i \leftarrow 1$, $e' \leftarrow e$, taken $\leftarrow$ FALSE.
  while $((i \leq k+1)$ and $e' \neq$ NULL$)$ **do**
  Set $(t', e'') \leftarrow$ INSERT-MSF$(F_i, e', p_{e'})$.
  if $(e' = e)$ **then** set taken $\leftarrow t'$ **endif**
  Set $e' \leftarrow e''$ and $i \leftarrow i+1$.
  endwhile
  if $(e' \neq e)$ **then return** (taken, e')
  else return (taken, NULL).

The boolean value that is returned indicates whether e belongs to F or not, the second value returns an edge that is removed from F, if any.

Recall that $F = \bigcup_{i \leq k+1} F_i$. We use the abbreviation NI-SPARSIFIER(k) to refer to this data-structure. Throughout the algorithm we will associate a weight with each edge in F and use F^w to refer to this weighted version of F.

Lemma 5.5.5. *For $k > 0$ and any graph G, NI-SPARSIFIER(k) maintains a weighted mfsd of order $k+1$ of G under edge insertions. The algorithm uses $O(kn)$ space and the total time for inserting m edges is $O(km \log n)$.*

Proof. We first show that NI-SPARSIFIER(k) maintains a forest decomposition such that (1) the forests are edge-disjoint and (2) each forest is maximal. We proceed by induction on the number m of edge insertions.

For $m = 0$, the forest decomposition is empty. Thus the edge-disjointness and maximality of forests trivially hold. For $m > 0$, consider the m-th edge insertion, which inserts an edge e. Let F', resp. F, denote the union of forests before, resp. after, the insertion of edge e. By the inductive assumption, F' satisfies (1) and (2). If $F = F'$, i.e., the edge e was not added to any of the forests when INSERT-NI(e, p_e) was called, then F also satisfies (1) and (2). Otherwise $F \neq F'$ and note that by construction, e is appended to exactly one forest. Let F'_j, resp. F_j, denote such maximal forest before, resp. after, the insertion of e. We distinguish two cases. If e links two trees of F'_j, then F_j is also a maximal forest and forests of F are edge-disjoint. Thus F satisfies (1) and (2). Otherwise, the addition of e results in the deletion of another edge $e' \in F'_j$. It follows that F_j is maximal and the current forests are edge-disjoint. Applying a similar argument to the addition of edge e' in the remaining forests, we conclude that F satisfies (1) and (2).

We next argue about time and space complexity. The dynamic tree data structure can be implemented in $O(n)$ space, where each query regarding max-weight$(F_i(u, v))$ can be answered in $O(\log n)$ time. Since the algorithm maintains $k + 1$ such forests, the space requirement is $O(kn)$. The total running time follows since insertion of an edge can result in at most $k + 1$ executions of the INSERT-MSF(F_i, e, p_e) procedures, each running in $O(\log n)$ time. □

(2) LIMITED EXACT MIN-CUT(k) data-structure: We use Algorithm 19 to implement the following operations for any unweighted graph G and parameter k,

- INSERT-LIMITED(e): Executes the insertion of edge e using Algorithm 19.
- QUERY-LIMITED$()$: Returns λ.
- INITIALIZE-LIMITED(G, k): Builds a data structure for G with parameter k by executing Step 1 of Algorithm 19 and then INSERT-LIMITED(e) for each edge e in G.

We use the abbreviation LIM(k) to refer to such data-structure. Combining the above data-structures leads to Algorithm 21.

Correctness and Running Time Analysis. Throughout the execution of Algorithm 21, F corresponds exactly to the msfd of order $k + 1$ of G maintained by NI-SPARSIFIER(k). In the following, let H be the graph that is given as input to LIM(k). Thus, by Corollary 5.5.1, QUERY-LIMITED$()$ returns $\min\{k, \lambda(H)\}$, i.e., it returns $\lambda(H)$ as long as $\lambda(H) \leq k$. We now formally prove the correctness.

Lemma 5.5.6. *Let $\epsilon \leq 1$, $k = 48 \log n/\epsilon^2$ and assume that the algorithm is started on an empty graph. As long as $\lambda(G) < k$, we have $H = G$, $p = k/4$, and QUERY-LIMITED$()$ returns $\lambda(G)$. The first rebuild step is triggered after the first insertion that increases $\lambda(G)$ to k and at that time, it holds that $\lambda(G) = \lambda(H) = k$.*

Algorithm 5.4: $(1+\epsilon)$-Min-Cut with $O(n \log n/\epsilon^2)$ space

```
1 Set k ← 48 log n/ε²
  Set p ← 12 log n/ε²
  Let H and F^w be empty graphs
2 Initialize-Limited(H, k)
  while Query-Limited() < k do
      Receive the next operation
      if it is a query then
          return Query-Limited()/ min{1, p}
      else if it is the insertion of an edge e then
          Sample a random weight from [0, 1] for the edge e and denote it by p_e
          if p_e ≤ p then
              Insert-Limited(e)
          Set (taken, e') ← Insert-NI(e, p_e)
          if taken then
              Insert e into F^w with weight p_e
              if e' ≠ NULL then
                  Remove e' from F^w
3 Set p ← p/2        // Rebuild Step
  Let H be the unweighted subgraph of F^w consisting of all edges of weight at most p
  Goto Step 2
```

Proof. The algorithm starts with an empty graph G, i.e., initially $\lambda(G) = 0$. Throughout the sequence of edge insertions $\lambda(G)$ never decreases. We show by induction on the number m of edge insertions that $H = G$ and $p = k/4$ as long as $\lambda(G) < k$.

Note that $k/4 \geq 1$ by our choice of ϵ. For $m = 0$, the graphs G and H are both empty graphs and p is set to $k/4$. For $m > 0$, consider the m-th edge insertion, which inserts an edge e. Let G and H denote the corresponding graphs after the insertion of e. By the inductive assumption, $p = k/4$ and $G \setminus \{e\} = H \setminus \{e\}$. As $p \geq 1$, e is added to H and, thus, it follows that $G = H$. Hence, $\lambda(H) = \lambda(G)$. If $\lambda(G) < k$ but $\lambda(G \setminus \{e\}) < k$, no rebuild is performed and p is not changed. If $\lambda(G) = k$, then the last insertion was exactly the insertion that increased $\lambda(G)$ from $k-1$ to k. As $H = G$ before the rebuild, Query-Limited() returns k, triggering the first execution of the rebuild step. □

We next analyze the case that $\lambda(G) \geq k$. In this case, both H and p are random variables, as they depend on the randomly chosen weights for the edges. Let $F(p)$ be the unweighted subgraph of F^w that contains all edges of weight at most p.

Lemma 5.5.7. *Let $N_h(p)$ be the graph consisting of all edges that were inserted after the last rebuild and have weight at most p and let $F^{old}(p)$ be $F(p)$ right after the last rebuild. Then it holds that $H = F^{old}(p) \cup N_h(p)$.*

Proof. Up to the first rebuild, $N_h = G$ and $p \geq 1$. Thus $N_h(p) = N_h = G$. Lemma 5.5.6 shows that until the first rebuild $H = G$. As $F^{\text{old}}(p) = \emptyset$, it follows that $H = G = N_h(p) \cup F^{\text{old}}(p)$ up to the first rebuild.

Immediately after each rebuild step, $N_h = \emptyset$ and H is set to be $F(p)$, thus the claim holds. After each subsequent edge insertion that does not trigger a rebuild, the newly inserted edge is added to $N_h(p)$ and to H iff its weight is at most p. Thus, both $N_h(p)$ and H change in the same way, which implies that $H = F^{\text{old}}(p) \cup N_h(p)$. □

Lemma 5.5.8. *At the time of a rebuild $F(p)$ is an msfd of order $k+1$ of $G(p)$.*

Proof. NI-SPARSIFIER maintains a maximal spanning forest decomposition based on minimum-weight spanning forests $F_1, \ldots F_{k+1}$ of G using the weights p_e. Now consider the hierarchical decomposition $F_1(p), \ldots, F_{k+1}(p)$ of $G(p)$ induced by taking only the edges of weight at most p of each forest F_i. Note that NI-SPARSIFIER would return exactly the same hierarchy $F_1(p), \ldots, F_{k+1}(p)$ if only the edges of $G(p)$ were inserted into NI-SPARSIFIER. Thus $F_1(p), \ldots, F_{k+1}(p)$ is an msfd of order $k+1$ of $G(p)$. □

In order to show that $\lambda(H)/\min\{1,p\}$ is an $(1+\epsilon)$-approximation of $\lambda(G)$ with high probability, we need to show that if $\lambda(G) \geq k$ then (a) the random variable p is at least $k/(4\lambda(G))$ w.h.p., which implies that $\lambda(G(p))$ is a $(1+\epsilon)$-approximation of $\lambda(G)$ w.h.p. and (b) that $\lambda(H) = \lambda(G(p))$ (by Lemma 5.5.4).

Lemma 5.5.9. *Let $\epsilon \leq 1$. If $\lambda(G) \geq k$, then (1) $p \geq k/(4\lambda(G))$ with probability $1 - O(\log n/n^4)$ and (2) $\lambda(H) = \lambda(G(p))$.*

Proof. For any $i \geq 0$, after the i-th rebuild we have $p = p^{(i)} := 12\log n/(2^i\epsilon^2)$. Let $\ell = \lfloor \log(12\log n/\epsilon^2) \rfloor$ denote the index of the last rebuild at which $p^{(i)} \geq 1$. For any $i \geq \ell + 1$, we will show by induction on i that (1) $p^{(i)} = 12\log n/(2^i\epsilon^2) \geq 12\log n/(\epsilon^2\lambda(G))$ with probability $1 - O((i-1-\ell)/n^4)$, which is equivalent to showing that $\lambda(G) \geq 2^i$ and that (2) at any point between the $(i-1)$-st and the i-th rebuild, $\lambda(H) = \lambda(G(p^{(i-1)}))$.

Once we have shown this, we can argue that the number of rebuild steps is small, thus giving the claimed probability in the lemma. Indeed, note that $\lambda(G) \leq n$ since G is unweighted. Additionally, from above we get that after the i-th rebuild, $\lambda(G) \geq 2^i$ with high probability. Combining these two bounds yields $i \leq O(\log n)$ w.h.p., i.e., the number of rebuild steps is at most $O(\log n)$.

We first analyse $i = \ell + 1$. Note that $\ell + 1$ is the index of the first rebuild at which $p^{(i)} < 1$. Assume that the insertion of some edge e caused the first rebuild. Lemma 5.5.6 showed that (1) at the first rebuild $\lambda(G) = k$ and (2) that up to the first rebuild $G(p) = G = H$. We observe that (1) and (2) remain true up to the $(\ell+1)$-st rebuild. In addition, $\lambda(G) = k \geq 24\log n/\epsilon^2 \geq 2^i$, which implies that $p^{(i)} \geq 1/2$. This shows the base case.

For the induction step ($i > \ell + 1$), we inductively assume that (1) at the $(i-1)$-st rebuild, $p^{(i-1)} \geq 12\log n/(\epsilon^2\lambda(G^{\text{old}}))$ with probability $1 - O((i-2-\ell)/n^4)$,

where G^{old} is the graph G right before the insertion that triggered the i-th rebuild (i.e., at the last point in time when QUERY-LIMITED() returned a value less than k), and (2) that $\lambda(H) = \lambda(G(p^{(i-2)}))$ at any time between the $(i-2)$-nd and the $(i-1)$-st rebuild. Let e be the edge whose insertion caused the i-th rebuild. Define $G^{\text{new}} = G^{\text{old}} \cup \{e\}$. By induction hypothesis, with probability $1 - O((i-2-\ell)/n^4)$, $p^{(i-1)} \geq 12 \log n/(\epsilon^2 \lambda(G^{\text{old}})) \geq 12 \log n/(\epsilon^2 \lambda(G^{\text{new}}))$ as $\lambda(G^{\text{old}}) \leq \lambda(G^{\text{new}})$. Thus, by Lemma 5.5.4, we get that $\lambda(G^{\text{new}}(p^{(i-1)}))/p^{(i-1)} \leq (1+\epsilon)\lambda(G^{\text{new}})$ with probability $1 - O(1/n^4)$. Applying an union bound, we get that the two previous statements hold simultaneously with probability $1 - O((i-1-\ell)/n^4)$.

We show below that $\lambda(G^{\text{new}}(p^{(i-1)})) = \lambda(H^{\text{new}})$, where H^{new} is the graph stored in LIM(k) right before the i-th rebuild. Thus, $\lambda(H^{\text{new}}) = k$, which implies that

$$\begin{aligned}\lambda(G^{\text{new}}(p^{(i-1)})) = k = 48 \log n/\epsilon^2 &\leq (1+\epsilon)\lambda(G^{\text{new}}) \cdot p^{(i-1)} \\ &= (1+\epsilon)\lambda(G^{\text{new}}) \cdot 12 \log n/(2^{i-1}\epsilon^2),\end{aligned}$$

with probability $1 - O((i-1-\ell)/n^4)$. This in turn implies that with probability $1 - O((i-1-\ell)/n^4)$, $\lambda(G^{\text{new}}) \geq 2^{i+1}/(1+\epsilon) \geq 2^i$ by our choice of ϵ.

It remains to show that $\lambda(G^{\text{new}}(p^{(i-1)})) = \lambda(H^{\text{new}})$. Note that this is a special case of (2), which claims that at any point between that $(i-1)$-st and the i-th rebuild $\lambda(H) = \lambda(G(p^{(i-1)}))$, where H and G are the current graphs. Thus, to complete the proof of the lemma it suffices to show (2).

As H is a subgraph of $G(p^{(i-1)})$, we know that $\lambda(G(p^{(i-1)})) \geq \lambda(H)$. Thus, we only need to show that $\lambda(G(p^{(i-1)})) \leq \lambda(H)$. Let G^{i-1}, resp. F^{i-1}, resp. H^{i-1}, be the graph G, resp. F, resp. H, right after rebuild $i-1$ and let N_h be the set of edges inserted since, i.e., $G = G^{(i-1)} \cup N_h$. As we showed in Lemma 5.5.7, $H = F^{i-1}(p^{(i-1)}) \cup N_h(p^{(i-1)})$. Thus, $H^{i-1} = F^{i-1}(p^{(i-1)})$. Additionally, by Lemma 5.5.8, $F^{i-1}(p^{(i-1)})$ is an msfd of order $k+1$ of $G^{i-1}(p^{(i-1)})$. Thus by Lemma 5.3.2, for every cut $(A, V \setminus A)$ of value at most k in H^{i-1}, $\lambda(A, H^{i-1}) = \lambda(F^{i-1}(p^{(i-1)}), A) = \lambda(A, G^{i-1}(p^{(i-1)}))$, where $\lambda(A, G) = |E_G(A, V \setminus A)|$. Now assume towards contradiction that $\lambda(G(p^{(i-1)})) > \lambda(H)$ and consider a minimum cut $(A, V \setminus A)$ in H, i.e., $\lambda(H) = \lambda(A, H)$. We know that at any time $k \geq \lambda(H)$. Thus $k \geq \lambda(H) = \lambda(A, H)$, which implies $k \geq \lambda(A, H^{i-1})$. By Lemma 5.3.2 it follows that $\lambda(A, H^{i-1}) = \lambda(A, G^{i-1}(p^{(i-1)}))$. Note that $H = H^{i-1} \cup N_h(p^{(i-1)})$ and $G(p^{(i-1)}) = G^{i-1}(p^{(i-1)}) \cup N_h(p^{(i-1)})$. Let x be the number of edges of $N_h(p^{(i-1)})$ that cross the cut $(A, V \setminus A)$. Then $\lambda(H) = \lambda(H, A) = \lambda(A, H^{i-1}) + x = \lambda(A, G^{i-1}(p^{(i-1)})) + x = \lambda(A, G(p^{(i-1)}))$, which contradicts the assumption that $\lambda(G(p^{(i-1)})) > \lambda(H)$. □

Since our algorithm is incremental and applies only to unweighted graphs, we know that there can be at most $O(n^2)$ edge insertions. The above lemma implies that for any current graph G, Algorithm 21 returns a $(1+\epsilon)$-approximation to a min-cut of G with probability $1 - O(\log n/n^4)$. Applying an union bound over $O(n^2)$ possible different graphs, gives that the probability that the algorithm does

not maintain a $(1+\epsilon)$-approximation is at most $O(\log n/n^2) = O(1/n)$. Thus, at any time we return a $(1+\epsilon)$-approximation with probability $1 - O(1/n)$.

Theorem 5.5.10. *There is an $O(n \log n/\epsilon^2)$ space randomized algorithm that processes a stream of edge insertions starting from an empty graph G and maintains a $(1+\epsilon)$-approximation to a min-cut of G with high probability. The total time for insertiong m edges is $O(m\alpha(n) \log^3 n/\epsilon^2)$ and queries can be answered in constant time.*

Proof. The space requirement is $O(n \log n/\epsilon^2)$ since at any point of time, the algorithm keeps H, F^w, LIM(k), and NI-SPARSIFIER (k), each of size at most $O(n \log n/\epsilon^2)$ (Corollary 5.5.1 and Lemma 5.5.5).

When Algorithm 21 executes a *Rebuild Step*, only the LIM(k) data-structure is rebuilt, but not NI-SPARSIFIER(k). During the whole algorithm m INSERT-NI operations are performed. Thus, by Lemma 5.5.5, the total time for all operations involving NI-SPARSIFIER(k) is $O(m \log^2 n/\epsilon^2)$.

It remains to analyze Steps 2 and 3. By Corollary 5.5.1, INITIALIZE-LIMITED(H, k) takes at most $O(m\alpha(n) \log^2 n/\epsilon^2)$ total time (Step 2). The running time of Step 3 is $O(m)$ as well. Since the number of *Rebuild Steps* is at most $O(\log n)$, it follows that the total time for all INITIALIZE-LIMITED(H, k) calls in Steps 2 and the total time of Step 3 throughout the execution of the algorithm is $O(m\alpha(n) \log^3 n/\epsilon^2)$.

We are left with analyzing the remaining part of Step 2. Each query operation executes one QUERY-LIMITED() operation, which takes constant time. Each insertion executes one INSERT-NI(e, p_e) operation, which takes amortized time $O(\log^2 n/\epsilon)$. We maintain the edges of F^w in a balanced binary tree so that each insertion and deletion takes $O(\log n)$ time. As there are m edge insertions the remaining part of Step 2 takes total time $O(m \log^2 n/\epsilon^2)$. Combining the above bounds gives the theorem. □

5.6 Conclusion

We obtained two new algorithms for the incremental (global) minimum cut problem in undirected, unweighted graphs. Our first algorithm maintains exactly the value of a minimum cut and has an $O(\log^3 n \log\log^2 n)$ amortized time per edge insertion and $O(1)$ query time. The main techniques behind this algorithm are (1) constructing a small sparsifier that preserves the non-trivial minimum cuts (2) incrementally maintaining the value of the minimum cut on the sparsifier and (3) employing periodical rebuilds whenever the maintained sparsifier is not valid for the current graph. While we believe the maintained sparsifier might prove useful to extend our algorithm to less restrictive settings, techniques in (2) and (3) crucially exploit the fact that the underlying data-structure supports edge insertions. An important problem is whether there is a fully-dynamic algorithm for exactly maintaining the value of

the minimum cut in sub-linear query and update time. Perhaps a good starting point is trying to come up with a deletions-only algorithm.

Our second result maintains a $(1+\epsilon)$-approximation to the value of a minimum cut in poly-logarithmic update time while using only $O(n \log n/\epsilon^2)$ space. The main idea behind our construction is to first maintain all minimum cuts up to a given threshold using small space and then apply the randomized sparsification result due to Karger [151]. It is an interesting direction to explore whether similar guarantees can be achieved in the fully-dynamic or decremental setting. In fact, even in the less general setting, that ignores the space requirement, it is not known whether there are decremental algorithms that maintain the value of the minimum cut up to a $(1+\epsilon)$ multiplicative factor in poly-logarithmic update and query time.

CHAPTER 6

Fast Incremental Algorithms via Local Sparsifiers

We show $n^{o(1)}$-approximation incremental algorithms with $n^{o(1)}$ *worst-case* update and query time on an undirected weighted n-node graph for many problems including all-pairs shortest paths, all-pairs max flow and min cut, multi-commodity concurrent flow, and uniform sparsest cut. By increasing the time to n^{ϵ} for any fixed $\epsilon > 0$ the approximation factors can be improved to polylog(n), and for all-pairs shortest paths to $O(1)$. For the all-pairs shortest paths problem, no previous algorithm with both $o(n)$ worst-case update and query time was known. For the other problems, even algorithms with both $o(n)$ *amortized* update and query time were not known.

As key to our result, we introduce a new notion of a sparsifier, called *local sparsifier*, for any graph property $\mathcal{P}$ and present a new general technique that converts any efficient construction algorithm for a local sparsifiers for $\mathcal{P}$ into an *incremental* algorithm for approximately maintaining $\mathcal{P}$. This technique connects several open problems between the fields of graph sparsifiers and dynamic graph algorithms, and leads to challenging new research questions for graph sparsifiers.

6.1 Introduction

In a recent study of the usage of graphs in practice [222] it was shown that real-world graphs are usually very large and more than half of the graphs in the survey change frequently, i.e., are dynamic. Due to the large size of these graphs, a dynamic algorithm needs to have *sublinear* time per operation to be useful for these applications. Another interesting finding of the study is that more than 2/3 of the graph computations are for "non-basic" graph problems, i.e., for problems for which no linear-time static algorithm is known such as all-pairs shortest paths and vari-

ous forms of graph partitioning. However, the current state-of-the art in dynamic graph algorithms is far from solving these "non-basic" graph problems in sublinear time, for many of them not even a dynamic algorithm better than recomputation from scratch is known. The reason for this "lack" of efficient dynamic algorithms became clear only recently: it has been shown that under certain, widely accepted assumptions maintaining the *exact* answer of many "non-basic" graph problems is not possible in sublinear time [4, 6, 78, 135]. Thus, to design dynamic algorithms for these problems, it is necessary to study *approximation* algorithms for them.

In this chapter, we study several "non-basic" graph problems including all-pairs shortest paths, all-pairs max flow (and min cuts), multi-commodity concurrent flow, and uniform sparsest cut (defined in Section 6.1). Despite an extensive research on dynamic all-pairs shortest paths [10, 11, 39, 40, 61, 220, 224, 250], no previous algorithms were known with $o(n)$ worst-case update and query time on a general graph with n nodes. For other problems where near-optimal time algorithms in the static setting are well-studied (for example, max flow and multi-commodity concurrent flow [158, 188, 210, 229, 231], uniform sparsest cut [22, 163, 188, 230, 236]) even algorithms with both $o(n)$ *amortized* update and query time were not known.

Our results. We show *incremental* approximation algorithms for the above problems. Incremental algorithms are data structures that maintain information about a graph property while the graph is modified by a sequence of edge insertions. Our algorithms significantly break the $o(n)$ bound by showing $n^{o(1)}$-approximation algorithms with $n^{o(1)}$ worst-case update time for all above problems. By increasing the time to n^{ϵ} for any constant $\epsilon > 0$, the approximation factors can be improved topolylog(n) and $O(1)$ for all-pairs shortest paths. The precise statement is as follows:

Theorem 6.1.1. *For any two parameters $r, \ell \geq 1$, there are incremental approximation algorithms on weighted (capacitated) undirected n-node graphs for the following problems (as defined in Table 6.1) with their corresponding guarantees:*

1. *All-pairs max flow and min cuts: $O(\log n)^{4\ell}$-approximation, $\tilde{O}(n^{2/(\ell+1)})$ worst-case update and query time.*
2. *All-pairs shortest paths: $(2r-1)^{\ell}$-approximation, $\tilde{O}(n^{2/(\ell+1)}n^{2/r})$ worst-case update and query time.*
3. *Multi-commodity concurrent flow: $O(\log n)^{8\ell}$-approximation, $\tilde{O}(n^{2/(\ell+1)})$ worst-case update time, and $\tilde{O}(k^2)$ query time when there are k commodity pairs in the query.*
4. *Uniform Sparsest Cut: $O(\log n)^{8\ell}$-approximation, $\tilde{O}(n^{2/(\ell+1)})$ worst-case update time $O(1)$ query time.*

All the above algorithms are randomized, except the all-pairs shortest paths algorithm, which is deterministic.

Dynamic problems	Query
All-pairs max flow	Given (s,t), return the value of max flow from s to t.
All-pairs shortest paths	Given (s,t), return the distance from s to t.
Multi-commodity concurrent flow	Given $\{(s_i,t_i,\mathbf{d}(i))\}_{i=1}^{k}$, return the value α where, concurrently for all i, s_i can send $\alpha\mathbf{d}(i)$ unit of flow to t_i.
Uniform Sparsest Cut	Return $\Phi_G = \min_{S\subset V} \frac{\mathrm{cap}_G(S,V\setminus S)}{\lvert S\rvert\cdot\lvert V\setminus S\rvert}$.

Table 6.1: List of dynamic problems and their corresponding query operation. For a weighted graph $G = (V,E,\mathbf{w})$, we have that $\mathrm{cap}_G(S,V\setminus S) = \sum_{(u,v)\in E,u\in S,v\notin S}\mathbf{w}(u,v)$.

Previous cut/flow algorithms. Despite the fact that all-pairs max flow and min cuts, multi-commodity concurrent flow, and uniform sparsest cut, are central problems in combinatorial optimization and have been extensively studied in the static setting, there are essentially no fast algorithms in the dynamic setting. Using previous techniques, it is possible to get dynamic algorithms with $\tilde{O}(1)$ worst-case update time and $\tilde{O}(n)$ query time under the assumption that the adversary is oblivious.[1] To the best of our knowledge, there is no previous algorithm with both $o(n)$ update and query time, even when we are content with only amortized guarantees.

Most closely related work to our work is the dynamic algorithm due to [67] for explicitly maintaining all the values of all-pairs min-cuts in $\tilde{O}(m^2)$ update time. For s-t max flow where s and t are fixed, there is an incremental algorithm with $O(n)$ amortized update time [122]. If we restrict to bipartite graphs with a certain specific structure, there is a $(1+\epsilon)$-approximation fully dynamic algorithm [12] with polylogarithmic worst-case update time. From the lower bound perspective, Dahlgaard [78] shows a conditional lower bound of $\Omega(n^{1-o(1)})$ amortized update time for exact incremental s-t max flow in *capacitated undirected* graphs. This shows that approximation is necessary to achieve sublinear running times.

Previous distance algorithms. The dynamic all pairs shortest paths problem has been extensively studied. Most previous work requires amortized update time. In particular, they either have $\Omega(n)$ update time or need to assume an oblivious adversary [11, 32, 39, 40, 61, 79, 134, 220]. An exception here is the work due to Alstrup et al. [18] that shows a very fast amortized deterministic algorithm for approximating the distance between two nodes, but this works *only if the queried distance are short.*

[1] We maintain a dynamic cut-sparsifier (against oblivious adversary) of size $\tilde{O}(n)$ due to [12] with $\tilde{O}(1)$ update time, and when given a query, we execute the fastest static approximation algorithms on the sparsifier in $\tilde{O}(n)$ time (using, for example, [210] for $(1+\epsilon)$-approximate max flow, [229] for $(1+\epsilon)$-approximate multi-commodity concurrent flow, and [230] for $O(\sqrt{\log n})$-approximate uniform sparsest cuts).

For worst-case update time, all previous algorithms [10, 224, 250] give *exact* answers but require $\Omega(n^{1.8})$ update time. If we allow a large approximation factor, then the best algorithm to our knowledge is an $O(\sqrt{\log n})$-approximation algorithm with $O(n^{1+o(1)})$ worst-case update time. This also assumes an oblivious adversary[2]. To summarize, our all-pairs shortest paths algorithm is the first algorithm with $o(n)$ worst-case update. Moreover, it is deterministic.

Even for the more restricted dynamic s-t shortest path problem, the story is similar as there is no $o(n)$ worst-case update algorithm. All previous amortized algorithms either take at least $\Omega(n^{3/4})$ on sparse graphs [38, 42, 43, 99] or assume an oblivious adversary [46, 133]. There is in fact, a conditional lower bound of $\Omega(n^{2-o(1)})$ worst-case update time for the incremental dynamic s-t exact shortest paths on weighted graphs [6, 135]. This again shows that approximation is necessary to obtain our worst-case update time.

Our techniques As a key to our results, we introduce a new notion of graph sparsifier, called *local sparsifier*. It is a stronger version of a well-studied notion called *vertex sparsifier* [60, 72, 94, 180, 191, 197]. Here, we give informal definitions. Let $G = (V, E)$ be a graph and, for each $u, v \in V$, let $\mathcal{P}(u, v, G)$ denote a *property* between u and v in G. For example, $\mathcal{P}(u, v, G)$ is the distance or the size of the u-v min cut. Let $K \subseteq V$ be a set of nodes called *terminals*. A *vertex sparsifier* of G with respect to K is a graph $H = (V', E')$ such that 1) $|V'| \approx |K|$ and 2) $\mathcal{P}(u, v, H) \approx \mathcal{P}(u, v, G)$ for all $u, v \in K$. That is, H has size close to K but still "approximately preserves" the property $\mathcal{P}$ between all terminal nodes.

A local sparsifier of G is a graph which contains possible vertex sparsifiers with respect to *any* given set of terminals. More precisely, a local sparsifier of G is a graph H such that, for any terminal set K, there is a subgraph of H, denoted by $H[K]$[3], where $H[K]$ is vertex sparsifier of G with respect to K. See Section 6.2 for the formal definition.

Our main technical contribution is a *meta-theorem* which turns any efficient construction for local sparsifiers for any property $\mathcal{P}$ into fast incremental algorithms for $\mathcal{P}$. Our reduction gives worst-case update time bounds and it is deterministic. Given a randomized sparsifier construction, the resulting incremental algorithm is also randomized. Details on this construction can be found in Section 6.3.

Given the meta-theorem, we then show that existing efficient constructions of vertex sparsifiers, such as algorithms for computing Räcke trees in [216] and Thorup-Zwick emulators in [218, 251], can be adapted to build local sparsifiers. Details on these constructions can be found in Sections 6.4 and 6.5. By plugging these constructions into our framework, we obtain our results on all-pairs max flow and all-pairs shortest paths Theorem 6.1.1. In fact, it is simple to extend our data-structure and

[2]They maintain a dynamic $O(\sqrt{\log n})$-spanner [45] (against oblivious adversary) of size $O(n^{1+o(1)})$ with $n^{o(1)}$ worst-case update time. Then, given a query, we run a static shortest path algorithm.

[3]This may not be a subgraph induced by K.

show an incremental algorithm for maintaining a *tree flow sparsifier* (i.e. Räcke tree [216]) itself.

Theorem 6.1.2 (Informal)**.** *For any $\ell \geq 1$, there is an incremental randomized algorithm with $\tilde{O}(n^{2/(\ell+1)})$ worst-case time for maintaining a tree flow sparsifier of an n-node graph G with quality $O(\log^{8\ell} n)$ and depth $O(\ell \log^2 n)$.*

See Section 6.6 for the formal definition of a tree flow sparsifier and its quality. Basically, it is a tree which "approximately preserves" all the cut/flow information of the graph. From Theorem 6.1.2, the simple structure of low-depth tree allows us to further implement other algorithms on the tree. Then, we easily obtain incremental algorithms for uniform sparsest cut and multi-commodity concurrent flow as stated Theorem 6.1.1.

Offline Fully Dynamic Algorithms. An *offline* dynamic algorithm is an algorithm where the whole sequences of updates (edge insertions and deletions) and queries is given as an input, and the algorithm needs to output information of the updated graph at every step that is queried. We say that an offline dynamic algorithm has (average) update and query time of t, if given a sequence of length L, then the total running time is $t \cdot L$.

Although the offline setting is a weaker than the standard dynamic setting, it is interesting for two reasons. First, offline algorithms are used to obtain fast static algorithms (e.g. [56, 181]). Second, many conditional lower bounds (e.g. [4, 6, 78]) for the standard dynamic setting also hold for the offline dynamic setting. Thus, giving an efficient algorithm for the offline dynamic setting shows that no such conditional lower bound is possible.

Simplifying the technique for incremental algorithms we can show an "offline" version of the meta-theorem which converts any efficient construction of vertex sparsifiers to an *offline fully dynamic* algorithm (see Section 6.7). Note that this version is incomparable with the previous one: an offline fully dynamic algorithm is incomparable to an online incremental algorithm. As this meta theorem only need vertex sparsifiers which are weaker than local sparsifiers, we immediately obtain the following.

Corollary 6.1.3 (Informal)**.** *There are offline fully dynamic approximation algorithms for the same problems with the same parameters as in Theorem 6.1.1*[4]*.*

In fact, there were previous several offline algorithms in the literature which are based on vertex sparsifiers. This includes the offline algorithms for minimum spanning trees [95], effective resistance [181], and 2/3-edge connectivity [211]. Our offline meta-theorem puts all their work into one framework: by just identifying the efficient construction of vertex sparsifier from each of these woks, their results can be immediately reproduced.

[4]To make sense of this, we in fact must replace "worst-case update time" with "average update time". There is also no concept of adversary in the offline setting.

New directions for sparsifiers. Apart from the new algorithms we devised in Theorem 6.1.1 and Corollary 6.1.3, we believe that our meta-theorems are valuable by themselves. They explicitly connect open problems of the two fields, namely dynamic algorithms and graph sparsifiers: any new upper or lower bounds is immediately transferred via them (see Section 6.8 for particularly interesting examples.) This connection also motivates the following research directions for constructing sparsifiers:

(1) *Trading size for quality*: If there exists a near-linear time construction of a vertex sparsifier with respect to terminals K which has size as large as $O(|K|n^{o(1)})$ but preserve a graph property within a factor of $(1+\epsilon)$ for any $\epsilon > 0$, then the resulting offline dynamic algorithms would have $n^{o(1)}$ update time and approximation factor only $(1+\epsilon')$ for any $\epsilon' > 0$. A similar implication holds for local sparsifier and incremental algorithms. This will give a significant improvement over our results that have large approximation factor. To the best of our knowledge, this question has not been explored in the vertex sparsifier literature since the research has concentrated on obtaining a vertex sparsifier whose size depends only on $|K|$. In fact, even a vertex sparsifier of size poly$(|K|n^{o(1)})$ with the $(1+\epsilon)$ factor would still give an interesting implication for dynamic algorithms (see e.g. Theorem 6.8.2).

(2) *Local sparsifier for effective resistance*: A near-linear time construction for vertex sparsifiers for effective resistance is known, i.e. an approximate Schur complement. This gives a very fast offline algorithm for effective resistance, as observed in [181]. However, in order to get an incremental algorithm, we would need a local sparsifier with an efficient construction. We are not aware whether such sparsifiers exist and we pose this as an important open question.

(3) *Speeding up existing constructions*: In this chapter, we only used existing sparsifiers that admit fast construction oracles. However, for example, there exist sparsifiers with better approximation quality for which no fast construction algorithm is known (e.g. [72, 94]). Thus, it is an interesting research question to develop faster algorithms for constructing them.

6.2 Local Sparsifiers

Let $G = (V, E)$ be graph. For any $u, v \in V$ we define $\mathcal{P}(u, v, G)$ to be a *property* between vertices u and v in G. Throughout $\mathcal{P}(u, v, G)$[5] will be a solution to a minimization problem involving u and v in G. We next review several notions that allows us to reduce the size of G while (approximately) retaining pair-wise information for some properties of G.

Definition 6.2.1 (Sparsifiers)**.** *Let $G = (V, E)$ be a graph, and let $\alpha \geq 1$. A graph $H = (V', E')$ with $V \subseteq V'$ is an α-sparsifier of G iff for every $u, v \in V$*

$$\mathcal{P}(u, v, G) \leq \mathcal{P}(u, v, H) \leq \alpha \cdot \mathcal{P}(u, v, H).$$

[5]Our idea extends also to other graph properties, but we decided to work with minimization problems in order to simplify the presentation.

The above notion captures different forms of sparsification. When $V' = V$ and $E' \subseteq E$, then H is referred to as *edge* sparsifier of G. Another example is when H contains additional vertices and edges which do not appear in G but H has a simpler structure than G.

The following sparsification notion is particularly useful if the goal is to reduce the vertex count of the input graph G.

Definition 6.2.2 (Vertex Sparsifiers)**.** *Let $G = (V, E)$ be a graph, with a terminal set $K \subseteq V$, and let $\alpha \geq 1$. A graph $H = (V', E')$ with $K \subseteq V'$ is an α-vertex sparsifier of G with respect to K iff for every $u, v \in K$*

$$\mathcal{P}(u, v, G) \leq \mathcal{P}(u, v, H) \leq \alpha \cdot \mathcal{P}(u, v, G).$$

Our work requires that sparsifiers satisfy two important properties, namely transitivity and decomposability. While transitivity is obvious, decomposability gives the following useful fact: if a graph is a combination of two graphs on disjoint edge sets, combining the respective sparsifiers of these graphs gives a sparsifier for the original graph. We next make these statement more precise.

Given a graph $G = (V, E)$, a parameter $\alpha \geq 1$, and an α-sparsifier H of G, we define S to be a mapping that takes G and α as inputs and produces H, i.e., $H := S(G, \alpha)$. We call such a mapping a *sparsifier mapping*. This leads to the following definition.

Definition 6.2.3 (Transitivity)**.** *Assume a sparsifier mapping S fulfills the following condition: For any graph G and parameters $\alpha_1 \geq 1$ and $\alpha_2 \geq 1$ it holds that when $H_1 = S(G, \alpha_1)$ and $H_2 = S(H_1, \alpha_2)$ then H_2 is an $\alpha_1\alpha_2$-sparsifier of G. Then we say that the mapping S is closed under transitivity.*

Definition 6.2.4 (Decomposability)**.** *Assume a sparsifier mapping S fulfills the following condition: For any two edge-disjoing graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ over a set V of nodes when $H_1 = S(G_1, \alpha)$ and $H_2 = S(G_2, \alpha_2)$ then $H = H_1 \cup H_2$ is an $\max\{\alpha_1, \alpha_2\}$-sparsifier of G. Then we say that S is closed under decomposition.*

We next introduce a new notion of sparsification that captures properties of both sparsifiers and vertex sparsifiers.

Definition 6.2.5 (Local Sparsifiers)**.** *Let $G = (V, E)$ be a graph and $\alpha_1 \geq 1$ be a parameter. A graph $H = (V', E')$ with $V \subseteq V'$ is a local sparsifier of G with quality $\alpha \geq 1$ iff the following hold:*

1. *The graph H is an α-sparsifier of G,*
2. *For every $K \subseteq V$, there exists a subgraph $H[K]$ of H such that $H[K]$ is α-vertex sparsifier of G with respect to K. Additionally, if K is a proper subset of V, then $H[K]$ must be a proper subgraph of H.*

In other words, the above definition suggests that local sparsifiers are sparsifiers from which we can extract vertex sparsifiers for any set of terminals K. Note that there are $\Theta(2^n)$ different terminal sets, thus Condition (2) of local sparsifiers is very strong. The transitivity and decomposability notions readily extend to local sparsifiers.

Since we will exploit local sparsifiers to speed up dynamic graph algorithms, it is natural to define some notion that involves running times for manipulating local sparsifiers. We address this in the following definition, where we introduce a data-structure version of local sparsifiers. To avoid overloading the notation, we will simply refer to this data-structure as local sparsifiers.

Definition 6.2.6. *Given a graph $G = (V, E)$ and a parameter $\alpha \geq 1$, a data-structure local sparsifier or simply a local sparsifier $H = (V', E')$ with quality α of G is a data-structure supporting the following operations:*

- *Preprocess(G, α): compute an α-sparsifier H of G,*
- *QuerySparsifier(G, K): compute the subgraph $H[K]$ of H and return $H[K]$ as an α-vertex sparsifier of G with respect to K.*

The above data-structure is characterized by two important measures: *preprocessing time*, which denotes the time for executing the operation Preprocess(G, α), and *query time*, which denotes the time for executing the operation QuerySparsifier(G, K). Note that the data-structures always produces a mapping S and and we will exploit properties of this mapping, specifically transitivity and decomposability, in our dynamic algorithms.

Since our goal is to design incremental algorithms with sub-linear update and query time, we will focus on building a local sparsifier with $\tilde{O}(m \cdot f(n))$ preprocessing time, while supporting queries in time $\tilde{O}(|K| \cdot g(n))$, where $f(n), g(n)$ are both sub-linear functions in n. In other words, this means that after computing a sparsifier of the input graph in time roughly proportional to its size, for any given set of terminals, we can construct a vertex sparsifier with respect to the terminals in time which depends *only* on the number of terminals, up to sub-linear factors. We make precise this requirement in the following definition.

Definition 6.2.7. *Let $G = (V, E)$ be a graph, and let $f(n), g(n) \geq 1$ be functions. We say that $(H, \alpha, f(n), g(n))$ is an efficient local sparsifier with quality $\alpha \geq 1$ of G iff H is a local sparsifier of quality α, and the preprocessing and query time of H are bounded by $O(m \cdot f(n))$ and $O(|K| \cdot g(n))$, respectively.*

To simplify the presentation, we will abuse the notation and sometimes write (H, α) instead of $(H, \alpha, f(n), g(n))$ when the runtime overheads are not important in specific contexts.

6.3 From Local Sparsifiers to Incremental Algorithms

In this section we show how to use efficient local sparsifiers to design online (approximate) incremental algorithms for problems with certain properties while achieving fast worst-case update and query time. Roughly speaking, the key idea behind our result is to form a set K out of the endpoints of all inserted edges since the last rebuild, and to use the efficient local sparsifiers with this set K to build a suitable vertex sparsifier at query time on which we answer the query using a static algorithm.

Theorem 6.3.1. *Let $G = (V, E)$ be a graph, and for any $u, v \in V$, let $\mathcal{P}(u, v, G)$ be a solution to a minimization problem between u and v in G. Let $f(n), g(n), h(n) \geq 1$ be functions, $\alpha, \ell \geq 1$ be parameters associated with the approximation factor, and let $\beta_0, \beta_1, \ldots, \beta_\ell$ with $\beta_0 = m$ be parameters associated with the running time. Assume the following properties are satisfied*

1. *G admits an efficient local sparsifier $(H, \alpha, f(n), g(n))$,*
2. *H is transitive and decomposable,*
3. *The property $\mathcal{P}(u, v, G)$ can be computed in $O(mh(n))$ time in a graph with m edges and n vertices.*

Then there is an incremental (approximate) dynamic algorithm that maintains for every pair of nodes u and v, an estimate $\delta(u, v)$, such that

$$\mathcal{P}(u, v, G) \leq \delta(u, v) \leq \alpha^\ell \cdot \mathcal{P}(u, v, G), \tag{6.1}$$

with worst-case update and query time of

$$\tilde{O}\left(\left(\sum_{j=1}^{\ell}\left(\frac{\beta_{j-1}}{\beta_j}\right) f(n) + \beta_\ell h(n)\right) g(n)\right) \quad \text{where } \beta_0 = m. \tag{6.2}$$

To gain some intuition, we first consider just a two-level scheme and then explain how this scheme naturally generalizes to more levels. Given an initial graph $G = (V, E)$ and an approximation parameter $\alpha \geq 1$, we build a data-structure that maintains

1. an efficient local sparsifier $(H, \alpha, f(n), g(n))$ of G (Theorem 6.3.1 Part 1), and
2. a set of edges E_1, which is initially set to empty.

Our data-structure is initialized using the Preprocess(G, α) operation of $(H, \alpha, f(n), g(n))$, and it is rebuilt every β_1 insertions, for some $\beta_1 \geq 0$ to be fixed later. Unless otherwise started, we will refer to G as the current graph. We next describe the Insert(e) and Query(s, t) operations. Upon insertion of a new edge e in G, we simply append edge e to E_1. For answering (s, t) queries, we first create the terminal set

$$K = \cup_{e \in E_1} V(e) \cup \{s, t\}, \tag{6.3}$$

where $V(e)$ are the endpoints of e, and then invoke QuerySparsifier(G, K) to get a vertex sparsifier $H[K]$ of $G \setminus E_1$ with respect to the terminal set K. Finally, we set $H' = H[K] \cup E_1$, and run on H' a static algorithms that computes property $\mathcal{P}(s, t, H)$ between s and t in H, denoted by $\delta_{H'}(s, t)$, and return this value as an estimate.

We next argue that the $\delta_{H'}(s, t)$ approximates property $\mathcal{P}(s, t, G)$ up to an α factor. Note that it is sufficient to show that H is an α-vertex sparsifier of G with respect to K. To this end, by definition of local sparsifiers, $H[K]$ is an α-vertex sparsifier of $G \setminus E_1$ with respect to K, which in turn implies that $H' = H[K] \cup E_1$, is an α-vertex sparsifier of $(G \setminus E_1) \cup E_1 = G$ with respect to K. The latter follows by decomposability of efficient local sparsifiers (Theorem 6.3.1 Part 2) and since endpoints of E_1 are added as terminals to K (Equation (6.3)).

We next analyze the update time. Note that the initialization time of our data-structure cost $O(mf(n))$ (Theorem 6.3.1 Part 1), and recall that our data-structure is rebuilt every β_1 operations. Thus, the amortized update time per insertion is $O(mf(n)\beta_1^{-1})$. For the query time, note that the size of the terminal set K at any time is $O(\beta_1)$. By Theorem 6.3.1 Part 1, we get that the the size of the sparsifier H of G is $O(\beta_1 g(n))$. Finally, the query time is bounded by $O(\beta_1 g(n)h(n))$ assuming that $P(u, v, H)$ can be computed in $O(|E(H)|h(|V|)$ time.

Combining the above bounds on the update and query time, we obtain the following trade-off

$$O\left(\left(\frac{m}{\beta_1}\right) f(n) + \beta_1 g(n)h(n)\right)$$

which in turn bounds the amortized update time and worst-case query time. The update time can be turned into a worst-case guarantee by a standard global rebuilding technique (see, for example, [115], Section 3.3.2).

We next explain the generalization of our approach to a multi-level hierarchy.

Data Structure. Consider some integer parameter $\ell \geq 1$ and parameters $\beta_0 \geq \ldots \geq \beta_\ell$, with $\beta_0 = m$. Our data structure maintains

1. a hierarchy of edge sets $\{E_i\}_{1 \leq i \leq \ell}$, each associated with the parameters $\{\beta_i\}_{1 \leq i \leq \ell}$,
2. a hierarchy of efficient local sparsifiers $\{(H_i, \alpha^{i+1})\}_{0 \leq i \leq \ell-1}$ for $\{G_i\}_{0 \leq i \leq \ell-1}$, where $G_0 = G$, and remaining G_i's are graphs that will be specified later,

We initialize our data-structure by constructing an efficient local sparsifier (H_0, α_0) for the initial graph $G_0 = G$ (Theorem 6.3.1 Part 1), and setting $H_i \leftarrow H_0$. We also set $E_i \leftarrow \emptyset$ for $1 \leq i \leq \ell$.

We note that $\{E_i\}_{1 \leq i \leq \ell}$ will change over the course of the algorithm, as we will shortly make precise. For $1 \leq i \leq \ell$, we will use $E_i^{(t)}$, when necessary, to denote the set E_i after the edge insertion at time t.

The hierarchy $\{E_i\}_{1 \leq i \leq \ell}$ keeps track of the inserted edges among different levels in our update sequence. Maintaining these edges will be useful when deciding to

periodically rebuild parts of our data-structure. These periodical rebuilds will allow us to strictly reduce the running time at the cost of paying a multiplicative increase which is proportional to the number of levels k in the hierarchy.

Handling Insertions. Consider the insertion of edge $e = (u, v)$ in G. We maintain a variable i that represents the level in the hierarchy (initially set to 1), and a boolean variable rebuild (initially set to FALSE) that determines whether a rebuild is triggered at some level of the hierarchy when processing the insertion of e. While $i \leq \ell$ and rebuild equals FALSE, we proceed as follows. We add e to E_i, and test whether the size of E_i exceeds β_i. If the latter holds, we set rebuild $\leftarrow$ TRUE, and distinguish two cases depending on whether $i = 1$ or $i \geq 2$.

If $i = 1$, we recompute from scratch an efficient local sparasifier (H, α) of the current graph G, set $G_0 \leftarrow G$. Moreover, we set $H_j \leftarrow H_0$ for $i \leq j \leq \ell - 1$, and let $E_j \leftarrow \emptyset$ for $1 \leq j \leq \ell$.

If $i \geq 2$, our goal will be to recompute efficient local sparsifier H_{i-1} at level $(i-1)$ in the hierarchy. To this end, we first define the graph

$$R_{i-1} := H_{i-2}[V(E_{i-1})] \cup E_{i-1},$$

where H_{i-2} is the efficient local sparsifier that we maintain at level $(i-2)$, and $V(E_{i-1})$ denotes the endpoints of the edges in E_{i-1}. In other words, R_{i-1} is obtained by taking the union over the edges stored at level $(i-1)$ and the vertex sparsifier $H_{i-2}[V(E_{i-1})]$ with respect to $V(E_{i-1})$ associated to the graph at level $(i-2)$. We then construct an efficient local sparsifier (R'_{i-1}, α) of R_{i-1}. The efficient local sparsifier H_{i-1} is updated using the following rule

$$H_{i-1} \leftarrow (H_{i-2} \setminus H_{i-2}[V(E_{i-1})]) \cup R'_{i-1}.$$

Finally, we update the efficient local sparsifiers in the levels $(i, \ldots, \ell - 1)$ by setting $H_j \leftarrow H_{i-1}$, for $i \leq j \leq \ell - 1$. We also let $E_j \leftarrow \emptyset$, for $i \leq j \leq \ell$, and increment i by 1. This algorithm is depicted in Figure 6.1.

Handling Queries. To answer the query for the approximate property $\mathcal{P}(u, v, G)$ between any pair of vertices s and t in G we proceed as follows. We first create a terminal set using the endpoints of the edges stored at the last level E_ℓ together with s and t, i.e.,

$$K = \cup_{e \in E_\ell} V(e) \cup \{s, t\},$$

where $V(e)$ are the endpoints of e. We then proceed by querying the vertex sparsifier $H_{\ell-1}[K]$ with respect to K, and union this with the maintained edge set E_ℓ, i.e., we define an auxiliary graph

$$H := H_{\ell-1}[K] \cup E_\ell.$$

Finally, we run the algorithm from Theorem 6.3.1 Part 3 on H to calculate the property $\mathcal{P}(u, v, H)$ between u and v in H, which we denote by $\delta_H(s, t)$, and return this value as an estimate.

Algorithm 6.1: INSERT$(e = (u, v))$

```
1  Set i ← 1
2  Set rebuild ← FALSE
3  Set E ← E ∪ {(u, v)}
4  while i ≤ ℓ and rebuild = FALSE do
5      E_i ← E_i ∪ {(u, v)}
6      if |E_i| > β_i then
7          rebuild ← TRUE
8          if i = 1 then
9              Set G_0 ← G
10             Compute an efficient local sparsifier (H_0, α) of G_0 (Theorem 6.3.1
                 Part 1)
11         else
12             Let R_{i−1} ← H_{i−2}[V(E_{i−1})] ∪ E_{i−1}
13             Compute efficient local sparsifier (R'_{i−1}, α) of R_{i−1} (Theorem 6.3.1
                 Part 1)
14             Set H_{i−1} ← (H_{i−2} \ H_{i−2}[V(E_{i−1})]) ∪ R'_{i−1}
15         Set H_j ← H_{i−1}, for i ≤ j ≤ ℓ − 1
16         Set E_j ← ∅ for i ≤ j ≤ ℓ
17     Set i ← i + 1
```

Algorithm 6.2: QUERY(s, t)

```
1  Set K ← ∪_{e∈E_ℓ} V(e) ∪ {s, t}
2  Set H ← H_{ℓ−1}[K] ∪ E_ℓ
3  Let δ_H(s, t) be the result obtained by the algorithm from Theorem 6.3.1 Part 3.
4  return δ_H(s, t).
```

Correctness. Let G be the current graph throughout the execution of the algorithm. We will show that as long as $|E_1| \leq \beta_1$, the efficient local sparsifier we maintain at level $(k-1)$ is sufficient to give a good approximation to the graph property $\mathcal{P}$ between any two pair of vertices from G. Note that whenever $|E_1| > \beta_1$, the entire data-structure is built from scratch, and in this case, the local sparsifier H_0 is already a good estimate for G.

To make the above statements precise, we need to introduce some useful notation. First, recall that for $1 \leq i \leq \ell$, we use $E_i^{(t)}$ to denote the set E_i after the edge insertion at time t in our algorithm. Let E be the set of inserted edges so far in our graph, i.e., $G = G_0 \cup E$, where G_0 is the initial graph from the last rebuild of the entire data-structure (Line 8 in Algorithm 6.1) or from the beginning of the algorithm. For each $e \in E$, we let τ_e be the index of the lowest level edge set in the current hierarchy $\{E_i\}_{1 \leq i \leq \ell}$ that contains e, i.e.,

$$\tau_e = \max\{j \in \{1, \ldots, \ell\} \mid e \in E_j\}.$$

This naturally induces a partitioning of E defined as follows

$$E = \cup_{1 \leq i \leq \ell} \tilde{E}_i, \quad \text{where } \tilde{E}_i = \{e \in E \mid \tau_e = i\} \text{ for } 1 \leq i \leq \ell.$$

Let $\{H_i\}_{0 \leq i \leq \ell-1}$ be the hierarchy of current efficient local sparsifier that our data structure maintains. Using the partitioning of E, we next show that each of these local sparsifiers maintains information for property $\mathcal{P}$ with respect to some edge sets in the partition, where the size of the edge set increases with the number of levels. In particular, this implies that the lowest-level local sparsifier $H_{\ell-1}$ will be a good estimate to property $\mathcal{P}$ in the current graph G. This approach is formally summarized in the following lemma.

Lemma 6.3.2. *The graph H_i at level i is an α^{i+1}-efficient local sparsifier of $G_0 \cup \left(E \setminus \cup_{i+1 \leq j \leq \ell} \tilde{E}_j\right)$ for $0 \leq i \leq \ell - 1$.*

Proof. We proceed by induction on the level i of the hierarchy. For the base case, i.e., $i = 0$, by construction H_0 is a α^{0+1}-efficient local sparsifier of $G_0 \cup (E \setminus \cup_{1 \leq j \leq \ell} \tilde{E}_i) = G_0$, and thus the claim holds.

Let H_i the efficient local sparsifier that our algorithm maintains at level $i > 0$. We want to show that H_i is an α^{i+1}-efficient local sparsifier of $G_0 \cup \left(E \setminus \cup_{i+1 \leq j \leq \ell} \tilde{E}_j\right)$. To this end, note that it suffices to prove that H_i is an α-efficient local sparsifier of $H_{i-1} \cup \tilde{E}_i$. We show this claim

- using the induction hypothesis on H_{i-1}, i.e., that H_{i-1} is an α^i-efficient local sparsifier of $G_0 \cup \left(E \setminus \cup_{i \leq j \leq \ell} \tilde{E}_j\right)$, and
- using the transitivity on H_i and $H_{i-1} \cup \tilde{E}_i$ (Theorem 6.3.1 Part 2).

We these two facts and the decomposability of efficient local sparsifiers (Theorem 6.3.1 Part 2) we get that that H_i is an α^{i+1}-efficient local sparsifier of

$$G_0 \cup \left(E \setminus \cup_{i \leq j \leq \ell} \tilde{E}_j\right) \cup \tilde{E}_i = G_0 \cup \left(E \setminus \cup_{i+1 \leq j \leq \ell} \tilde{E}_j\right).$$

Thus it remains to show that H_i is an α-efficient local sparsifier of $H_{i-1} \cup \tilde{E}_i$. We distinguish two cases. (1) If $\tilde{E}_i = \emptyset$, then we know that there was a rebuild at a level smaller than i in the hierarchy, which implies that $H_i = H_{i-1}$ (Line 14 of Algorithm 6.1). Thus H_i is trivially an α-efficient local sparsifier of $H_{i-1} \cup \tilde{E}_i$. (2) If $\tilde{E}_i \neq \emptyset$, let t_i be the last time that H_i was rebuilt with respect to the set E_i, i.e., Lines 11-14 in Algorithm 6.1 were executed at time t_i. We claim that $\tilde{E}_i = E_i^{(t_i)}$. Note that this follows by definition of $\tilde{E}_i$ since edges belonging to this set do not appear in the levels larger than i. To prove the claimed approximation guarantee on H_i, we first observe that the graph $H_{i-1} \cup \tilde{E}_i$ can be partitioned into edge-disjoint graphs as follows

$$H_{i-1} \cup \tilde{E}_i = \left(H_{i-1} \setminus H_{i-1}[V(\tilde{E}_i)]\right) \cup \left(H_{i-1}[V(\tilde{E}_i)] \cup \tilde{E}_i\right),$$

where $R_i = H_{i-1}[V(\tilde{E}_i)] \cup \tilde{E}_i$ by our construction. Let (R'_i, α) be the efficient local sparsifier of R_i computed by the algorithm. By definition of efficient local sparsifiers we know that $V(R_i) \subseteq V(R'_i)$ and R'_i is an α-sparsifier of R_i. Moreover, recall that the algorithm updates H_i as follows

$$H_i = \Big(H_{i-1} \setminus H_{i-1}[V(\tilde{E}_i)]\Big) \cup R'_i.$$

Applying the decomposability property of Theorem 6.3.1 Part 2 on $H_{i-1} \setminus H_{i-1}[V(\tilde{E}_i)]$ and R'_i we get that H_i is an α-efficient local sparsifier of $H_{i-1} \cup \tilde{E}_i$, which completes the proof. □

We finally show that the estimate $\delta_H(s,t)$ returned by the query algorithm in Figure 6.2 approximates the property $\mathcal{P}$ of the current graph G up to an α^ℓ factor, thus proving the claimed estimate in Theorem 6.3.1.

By Lemma 6.3.2, we get that $H_{\ell-1}$ is an α^ℓ-efficient local sparsifier of graph $G_0 \cup (E \setminus \tilde{E}_\ell)$. Since $\tilde{E}_\ell = E_\ell$ (because ℓ is largest level), we get that $H_{\ell-1}[K]$ is a α^ℓ-vertex sparsifier of $G_0 \cup (E \setminus E_\ell)$ with respect to K. Using decomposability of efficient local sparsifiers (Theorem 6.3.1 Part 2), the latter implies that $H = H_{\ell-1} \cup E_\ell$ is a α^ℓ-vertex sparsifier of $G_0 \cup (E \setminus E_\ell) \cup E_\ell = G_0 \cup E = G$.

Running Time. We first study the update time of our data structure. To this end, it will be useful to bound the size of each efficient local sparsifier in the hierarchy $\{H_i\}_{0 \le i \le \ell-1}$ at any given point of time.

Lemma 6.3.3. *At any point of time, for each $0 \le i \le (\ell-1)$ and $K \subseteq V$, we have that*

$$|H_i[K]| \le \tilde{O}\left(|K| \cdot g(n)\right).$$

Proof. We actually prove something stronger, namely that at any point of time, for each $K \subseteq V$ and $0 \le i \le (\ell-1)$ we have that $|H_i[K]| \le O\left((i+1)|K| \cdot g(n)\right)$. As we will shortly see, the number of levels ℓ in the hierarchy does not exceed $O(\log n)$. Since $i \le (\ell-1)$, we immediately get the claimed bound of the lemma.

At any point of time during the execution of our data-structure, note that the worst-case bound on the size of $H_i[K]$ at level i is attained when the H_j for $0 \le j \le i$ are different i.e., each of the efficient local sparsifier has undergone a rebuild with respect to the current edge set E_j. Thus, throughout we assume that this is indeed the case, as otherwise the bounds can only get better.

We now prove the claim by induction on the level i of the hierarchy. For the base case, i.e., $i = 0$, we know that H_0 is an efficient local sparsifier of G_0, and by querying H_0 with respect to K it follows that $|H_0[K]| \le O(|K| \cdot g(n))$, and hence the claim holds.

By induction hypothesis we get that for each $K \subseteq V$, it holds that $|H_{i-1}[K]| \le O(i|K| \cdot g(n))$. We now show the inductive step. Let $K \subseteq V$ be any subset of vertices. To this end, let H_i be the efficient local sparsifier that has undergone a rebuild with respect to E_i at level $i > 0$. Let $R_i := H_{i-1}[V(E_i)] \cup E_i$ be the

intermediate graph which is used to rebuild H_i, and let (R'_i, α) be the efficient local sparsifier of R_i, as defined in Algorithm 6.1. Define $K' := K \cap V(R_i)$ and note that by construction $|V(R_i)| \leq n$. Then by querying R'_i with respect to K' we get that

$$|R'_i(K')| \leq O(|K'| \cdot g(|V(R_i)|)) \leq O(|K| \cdot g(n)).$$

Finally, since H_i is formed by taking the union of R'_i with some part of H_{i-1} we get that

$$|H_i| \leq |H_{i-1} \cup R'_i| \leq O((i+1)|K| \cdot g(n)),$$

where the bound on H_{i-1} follows by induction hypothesis.

□

The lemma below bounds the amortized update time of our data-structure.

Lemma 6.3.4. *The amortized time of* INSERT$(e = (u, v))$ *operation is bounded by*

$$\tilde{O}\left(\left(\sum_{j=0}^{\ell-1} \frac{\beta_j}{\beta_{j+1}}\right) f(n)g(n)\right).$$

Proof. For $0 \leq i \leq \ell - 1$, let $Y(i)$ be the amortized update time aggregated up to (and including) level i in the hierarchy. Furthermore, let $Z(i)$ be the amortized update time at level i in the hierarchy (and excluding all other levels). We will show by induction on the number of levels i that $Y(i) = \tilde{O}\left(\left(\sum_{j=0}^{i} \frac{\beta_j}{\beta_{j+1}}\right) f(n)g(n)\right)$, which with $i = (\ell - 1)$ implies the claimed bound of the lemma.

For the base case, i.e., $i = 0$, recall that the cost for constructing an efficient local sparsifier H_0 of the current graph G_0 is $\tilde{O}(\beta_0 f(n))$ (Theorem 6.3.1 Part 1), where $\beta_0 = m$ is the current number of edges. Moreover, the cost for updating the efficient local sparsifiers in the levels below $\{H_j\}_{1 \leq j \leq k-1}$ is bounded by $\tilde{O}(\ell\beta_0 f(n)) = \tilde{O}(\beta_0 f(n))$. Thus, the overall cost of a rebuild at level $i = 0$ is $\tilde{O}(\beta_0 f(n))$. Since H_0 is rebuilt every β_1 insertions, we get that the amortized cost per insertion is $Y(0) = Z(0) = \tilde{O}\left(\left(\frac{\beta_0}{\beta_1}\right) f(n)\right) = \tilde{O}\left(\left(\frac{\beta_0}{\beta_1}\right) f(n)g(n)\right)$, as desired.

We next show the inductive step. Consider the maintained efficient local sparsifier H_i at level i that undergoes a rebuild with respect to E_i, and let H_{i-1} be the efficient local sparsifier one level above (recall that a rebuild at level i is triggered by level $(i+1)$, i.e., because $|E_{i+1}| > \beta_{i+1}$). We want to bound the size of the intermediate graph $R_i = H_{i-1}[V(E_i)] \cup E_i$, as defined in Algorithm 6.1, which in turn determines the cost for rebuilding H_i. To this end, first observe that by construction $|E_i| \leq \beta_i$. Second, by Lemma 6.3.3 we get that

$$|H_{i-1}[V(E_i)]| \leq \tilde{O}(|V(E_i)|g(n)) \leq \tilde{O}(\beta_i g(n)).$$

Combining these two bounds we get that $|R_i| \leq \tilde{O}(\beta_i g(n))$. We now bound the cost for computing R'_i and updating the efficient local sparsifier H_i. As Algorithm 6.1 computes an efficient local sparsifier (R'_i, α) of R_i, by Theorem 6.3.1 Part 1

we get that the cost for computing R'_i is

$$\tilde{O}(|R_i|f(n)) = \tilde{O}(\beta_i g(n)) \cdot \tilde{O}(f(n)) = \tilde{O}(\beta_i f(n)g(n)).$$

Consider the update of the efficient local sparsifier H_i, and assume that before the update, $H_i = H_{i-1}$ holds. Then we can simply update H_i by deleting the edges $H_{i-1}[V(E_i)]$ from H_i and adding the new edges R'_i to H_i. Since by the above discussion the size of both $H_{i-1}[V(E_i)]$ and R'_i is bounded by $\tilde{O}(\beta_i f(n)g(n))$, we claim the cost for updating H_i is also bounded by $\tilde{O}(\beta_i f(n)g(n))$.

Now, if $H_i \neq H_{i-1}$ holds before updating H_i, this means that H_i has undergone already a rebuild with respect to E_i. We then reverse all the operations of the data-structure during the last rebuild until $H_i = H_{i-1}$, and proceed as above for updating H_i. Since by construction $|E_i| \leq \beta_i$, observe that the reversing cost cannot exceed the cost of updating H_i, which we showed to be at most $\tilde{O}(\beta_i f(n)g(n))$. Similarly, for updating the efficient local sparsifiers $\{H_j\}_{i-1 \leq j \leq \ell-1}$, we first reverse their the data-structure operations until $H_{i-1} = H_{i+1} = \ldots = H_{\ell-1}$, and then proceed as above for updating $\{H_j\}_{i-1 \leq j \leq \ell-1}$[6]. Since there are at most ℓ levels below to update during the rebuild at level i, the total cost for updating the hierarchy $\{H_j\}_{i \leq j \leq \ell-1}$ is bounded by

$$\tilde{O}(\ell\beta_i f(n)g(n)) = \tilde{O}(\beta_i f(n)g(n)).$$

Summing the cost for computing R'_i and the cost for updating the hierarchy $\{H_j\}_{i \leq j \leq \ell-1}$, we conclude that the total cost for rebuilding H_i with respect to E_i is bounded by $\tilde{O}(\beta_i f(n)g(n))$. Since the emulator H_i is rebuilt every β_{i+1} operations, we get that the amortized cost per operation is

$$Z(i) = \tilde{O}\left(\left(\frac{\beta_i}{\beta_{i+1}}\right) f(n)g(n)\right).$$

To complete the inductive step, note that by induction hypothesis

$$Y(i-1) = \tilde{O}\left(\left(\sum_{j=0}^{i-1} \frac{\beta_j}{\beta_{j+1}}\right) f(n)g(n)\right).$$

Summing over this and the bound on $Z(i)$ we get

$$\begin{aligned} Y(i) &= Y(i-1) + Z(i) \\ &= \tilde{O}\left(\left(\sum_{j=0}^{i-1} \frac{\beta_j}{\beta_{j+1}}\right) f(n)g(n)\right) + \tilde{O}\left(\left(\frac{\beta_i}{\beta_{i+1}}\right) f(n)g(n)\right) \\ &= \tilde{O}\left(\left(\sum_{j=0}^{i} \frac{\beta_j}{\beta_{j+1}}\right) f(n)g(n)\right). \end{aligned}$$

□

[6]Note that this is faster than copying H_{i-1} into the data structures for $\{H_j\}_{i-1 \leq j \leq \ell-1}$

We next study the query time of our data-structure.

Lemma 6.3.5. *The time for a* Query(s,t) *operation is bounded by* $\tilde{O}\left(\beta_\ell g(n)h(n)\right)$.

Proof. Let $K = \cup_{e\in E_\ell} V(e) \cup \{s,t\}$ be the set of terminals defined in Algorithm 6.2. By construction, we know that $|E_\ell| \leq \beta_\ell$, which in turn implies that $|K| \leq O(\beta_\ell)$. Let $H = H_{\ell-1}[K] \cup E_\ell$ be the graph estimator as defined in Algorithm 6.2, where $H_{\ell-1}$ is the efficient local sparsifier at level $(\ell-1)$ in the hierarchy. By Lemma 6.3.3 and the bound on the size of T, we get that $|H_{\ell-1}[K]| \leq \tilde{O}(\beta_\ell g(n))$, which in turn implies that

$$|H| = |H_{k-1}[K] \cup \beta_\ell| \leq \tilde{O}(\beta_\ell g(n)).$$

Since the algorithm for testing property $\mathcal{P}(s,t,G)$ runs in $\tilde{O}(|H|h(n))$ time by Theorem 6.3.1 Part 3, we get the our query time is bounded by $\tilde{O}(\beta_\ell g(n)h(n))$. □

Combining the bounds on the update and query time from Lemmas 6.3.4 and 6.3.5, we obtain the following trade-off

$$\tilde{O}\left(\left(\sum_{j=0}^{\ell-1}\left(\frac{\beta_j}{\beta_{j+1}}\right) f(n) + \beta_\ell h(n)\right) g(n)\right), \quad \text{where } \beta_0 = m,$$

which in turn proves the claimed update and query time in Theorem 6.3.1.

Finally we show for what choice of parameters $\{\beta_i\}_{0\leq i\leq \ell}$ the above trade-off is minimized, if we ignore functions $f(n), g(n)$ and $h(n)$. As we will see in the subsequent sections, this simplification will be justified in all the applications of Theorem 6.3.1.

Lemma 6.3.6. *For* $1 \leq \ell \leq \log n$, *let* $\{\beta_i\}_{0\leq i\leq \ell}$ *be a family of parameters with* $\beta_0 = m$. *If we set*

$$\beta_i = (\beta_{i-1})^{\frac{\ell-(i-1)}{\ell+1-(i-1)}},\ 1 \leq i \leq \ell$$

then

$$\tilde{O}\left(\sum_{j=0}^{\ell-1}\frac{\beta_j}{\beta_{j+1}} + \beta_\ell\right) = \tilde{O}\left(m^{1/k+1}\right).$$

Proof. We claim that for each $i \geq 1$, it holds that $\beta_i = m^{1-\frac{i}{\ell+1}}$, and prove this by induction on i. For the base case, i.e., $i = 1$, by the choice of β_1 we have $\beta_1 = (\beta_0)^{\frac{\ell}{\ell+1}} = m^{1-\frac{1}{\ell+1}}$.

For the inductive step, we have

$$\begin{aligned}\beta_i = (\beta_{i-1})^{\frac{\ell-(i-1)}{\ell+1-(i-1)}} &= \left(m^{1-\frac{(i-1)}{\ell+1}}\right)^{\frac{\ell-(i-1)}{\ell+1-(i-1)}} = m^{\frac{\ell+1-(i-1)}{\ell+1}\cdot\frac{\ell-(i-1)}{\ell+1-(i-1)}} \\ &= m^{1-\frac{i}{\ell+1}},\end{aligned} \tag{6.4}$$

where the second equality follows by induction hypothesis on β_{i-1}.

Plugging the choice of β_i in Equation 6.4 yields

$$\tilde{O}\left(\sum_{j=0}^{\ell-1}\frac{m^{1-\frac{j}{\ell+1}}}{m^{1-\frac{(j+1)}{\ell+1}}}+m^{\frac{1}{k+1}}\right)=\tilde{O}\left(\sum_{j=0}^{\ell-1}m^{\frac{1}{\ell+1}}+m^{\frac{1}{\ell+1}}\right)=\tilde{O}\left(\ell m^{\frac{1}{\ell+1}}\right)$$
$$=\tilde{O}\left(m^{\frac{1}{\ell+1}}\right).$$

□

6.4 Incremental All Pair Shortest Paths

In this section we show how to use our general Theorem 6.3.1 to design online incremental algorithms for the approximate All-Pair Shortest Path Problem with fast worst-case update and query time. Concretely, we will show that that assumptions (1) and (2) in Theorem 6.3.1 are satisfied with certain parameters for shortest paths. Note that (3) follows immediately by any $\tilde{O}(m)$ time single pair shortest path algorithm. This results in the following theorem.

Theorem 6.4.1. *Let $G=(V,E)$ be an undirected, weighted graph. For every $r,\ell\geq 1$, there is a deterministic incremental approximate APSP algorithm that maintains for every pair of nodes u and v, a distance estimate $\delta(u,v)$ such that*

$$dist_G(u,v)\leq\delta(u,v)\leq(2r-1)^{\ell}dist_G(u,v),$$

with worst-case update and query time of

$$\tilde{O}(n^{2/(\ell+1)}n^{2/r}).$$

We start by introducing the usual definitions of sparsifiers and vertex sparsifiers for distances. Having defined these, the definition of local sparsifiers becomes apparent from the general definition we introduced in Section 6.2. Let $G=(V,E)$ be an undirected, weighted graph with a *terminal* set $K\subseteq V$. For $u,v\in V$, let $\text{dist}_G(u,v)$ denote the length of a shortest path between u and v in G.

Definition 6.4.2 (Sparsifiers for Distances)**.** *Let $G=(V,E)$ be an undirected, weighted graph, and let $\alpha\geq 1$ be a stretch parameter. A graph $H=(V',E')$ with $V\subseteq V'$ is an α-sparsifier of G iff for all $u,v\in V$,*

$$dist_G(u,v)\leq dist_H(u,v)\leq\alpha\cdot dist_G(u,v).$$

Definition 6.4.3 (Vertex Sparsifiers for Distances)**.** *Let $G=(V,E)$ be an undirected, weighted graph with a terminal set $K\subseteq V$, and let $\alpha\geq 1$. A graph $H=(V',E')$ with $K\subseteq V'$ is an α-(vertex) distance sparsifier of G with respect to K iff for all $u,v\in K$,*

$$dist_G(u,v)\leq dist_H(u,v)\leq\alpha\cdot dist_G(u,v).$$

Algorithm 6.3: HierarchyConstruct(G, r)

$A_0 \leftarrow V$; $A_r \leftarrow \emptyset$
for $i \leftarrow 1$ *to* $r-1$ **do**
  $A_i \leftarrow \textsc{Sample}\left(A_{i-1}, |V|^{-1/r}\right)$
for *every* $v \in V$ **do**
  for $i \leftarrow 0$ *to* $r-1$ **do**
    Let $\text{dist}_G(A_i, v) \leftarrow \min\{\text{dist}_G(w, v) \mid w \in A_i\}$
    Let $p_i(v) \in A_i$ be such that $\text{dist}_G(p_i(v), v) = \text{dist}_G(A_i, v)$
  $\text{dist}_G(A_r, v) \leftarrow \infty$
  Let $B(v) \leftarrow \cup_{i=0}^{r-1}\{w \in A_i \setminus A_{i+1} \mid \text{dist}_G(w, v) < \text{dist}_G(A_{i+1}, v)\}$

We next show that the distance property in graphs admits efficient local sparsifiers. We achieve this by showing a deterministic variant of the distance oracle due to Thorup and Zwick [251]. While we closely follow the ideas presented in the deterministic oracle due to Roddity, Thorup and Zwick [218], we note that they only give a bound on the *total* size of the oracle, which is not sufficient for our purposes.

Lemma 6.4.4 (Efficient Distance Local Sparsifiers)**.** *Given an undirected, weighted graph $G = (V, E)$, and a parameter $r \geq 1$, there is a deterministic algorithm for constructing an efficient distance local sparsifier with $(2r-1)$ stretch, $\tilde{O}(mn^{1/r})$ preprocessing time, and $O(|K|n^{1/r})$ query time, where K is any set of queried terminals.*

We start by reviewing the randomized algorithm for APSP due to Thorup and Zwick [251](which is depicted in Figure 6.3), and then derandomize that algorithm and show how it can be used to solve the above problem.

1. Set $A = V$ and $A_r = \emptyset$, and for $1 \leq i \leq r-1$ obtain A_i by picking each node from A_{i-1} independently, with probability $n^{-1/r}$.
2. For each $1 \leq i < r$, and for each vertex $v \in V$, find the vertex $p_i(v) \in A_i$ (also known as the i-th *pivot*) that minimizes the distance to v, i.e.,

$$p_i(v) := \arg\min_{u \in A_i} \text{dist}_G(u, v),$$

   and its corresponding distance value

$$\text{dist}_G(A_i, v) := \min\{\delta(w, v) \mid w \in A_i\} = \text{dist}_G(v, p_i(v)).$$

3. For each vertex $v \in V$, define the *bunch* $B(v) = \cup_{i=0}^{r-1} B_i(v)$, where

$$B_i(v) := \{w \in A_i \setminus A_{i+1} \mid \text{dist}_G(w, v) < \text{dist}_G(A_{i+1}, v)\}.$$

Thorup and Zwick [251] showed that using the hierarchy of sets $(A_i)_{0 \leq i \leq r}$ chosen as above, the expected size of a bunch $\mathbb{E}\left[|B(v)|\right]$ is $O(rn^{1/r})$, for each vertex

$v \in V$. We note that the only place where their construction uses randomization is when building the hierarchy of sets (the **for** loop in Step 2 in Figure 6.3). Therefore, to derandomize their algorithm it suffices to design a deterministic algorithm that efficiently computes a hierarchy of set $(A_i)_{0 \leq i \leq r}$ such that $|B(v)| \leq \tilde{O}(rn^{1/r})$, for each $v \in V$ (note that compared to the randomized construction, we are content with additional poly-log factors on the size of the bunches).

We present a deterministic algorithm for computing the hierarchy of sets that closely follows the ideas presented in the deterministic construction of Roditty, Thorup, and Zwick [218]. The main two ingredients of the algorithm are the *hitting set* problem, and the *source detection* problem. For the sake of completeness, we next review their definitions and properties.

Definition 6.4.5 (Hitting set)**.** *Let U be a set of elements, and let $\mathbb{S} = \{S_1, \ldots, S_p\}$ be a collection of subsets of U. We say that T is a hitting set of U with respect to $\mathbb{S}$ if $T \subseteq U$, and T has a non-empty intersection with every set of $\mathbb{S}$, i.e., $T \cap S_i \neq \emptyset$ for every $1 \leq i \leq p$.*

It is known that computing a hitting set of minimum size is an NP-hard problem. In our setting however, it is sufficient to compute approximate hitting sets. Since our goal is to design a deterministic algorithm, one way to deterministically compute such sets is using a variant of the well-known greedy approximation algorithm: (1) Form the set T by repeatedly adding to T elements of U that 'hit' as many 'unhit' sets as possible, until only $|U|/s$ sets are unhit, where $|S_i| \geq s$ for each $1 \leq i \leq p$; (2) add an element from each one of the unhit sets to T. The lemma below shows that this algorithm finds a reasonably sized hitting set in time linear in the size of U the collection $\mathbb{S}$.

Lemma 6.4.6. *Let U be a set of size u and let $\mathbb{S} = \{S_1, \ldots, S_p\}$ be the collection of subset of U, each of size at least s, where $s \leq p$. Then the above deterministic greedy algorithm runs in $O(u + ps)$ time and finds a hitting set T of U with respect to $\mathbb{S}$, whose size is bounded by $|T| = (u/s)(1 + \ln p)$.*

Note that the size of this hitting set is within $O(\log n)$ of the optimum size since in the worst case T has size at least u/s.

Definition 6.4.7 (Source Detection)**.** *Let $G = (V, E)$ be an undirected, weighted graph, let $U \subseteq V$ be an arbitrary set of sources of size u, and let q be a parameter with $1 \leq q \leq u$. For every $v \in V$, we let $U(v, q, G)$ be the set of the q vertices of U that are closest to v in G.*

Roditty, Thorup, and Zwick [218] showed that the set $U(v, q, G)$ can be computed using q single-source shortest path computations. We review their result in the lemma below.

Lemma 6.4.8 ([218])**.** *For every $v \in V$, the set $U(v, q, G)$ can be computed in time $O(qm \log n)$.*

Algorithm 6.4: DETHIERARHCY(G, r)

Input : Undirected, weighted graph $G = (V, E)$, parameter $r \geq 1$
Output : Hierarchy of sets $(A_i)_{0 \leq i \leq r}$

$q \leftarrow \lceil n^{1/r}(1 + \ln n) \rceil$
$A_0 \leftarrow V$; $A_r \leftarrow \emptyset$
for $i \leftarrow 0$ *to* $r - 2$ **do**
  Compute $A_i(v, q, G)$ for each $v \in V$ using the source detection algorithm (Lemma 6.4.8)
  Let $\{A_i(v, q, G)\}_{v \in V}$ be the resulting collection of sets
  Compute a hitting set $A_{i+1} \subseteq A_i$ with respect to $\{A_i(v, q, G)\}_{v \in V}$ (Lemma 6.4.6)
return $(A_i)_{0 \leq i \leq r}$

Our algorithm for constructing the hierarchy of sets $(A_i)_{0 \leq i \leq r}$, depicted in Figure 25, is as follows. Initially, we set $A_0 = V$ and $A_r = \emptyset$. To construct the set A_{i+1}, given the set A_i, for $0 \leq i \leq p - 2$, we first find the set $A_i(v, q, G)$, where $q = \tilde{O}(n^{1/r})$, using the source detection algorithm from Lemma 6.4.8. Then we observe that the collection of sets $\{A_i(v, q, G)\}_{v \in V}$ can be viewed as an instance of the minimum hitting set problem over the set (universe) A_i, i.e., we want to find a set $A_{i+1} \subseteq A_i$ of minimum size such that each set $A_i(v, q, G)$ in the collection contains at least one node of A_{i+1}. We construct A_{i+1} by invoking the deterministic greedy algorithm from Lemma 6.4.6, which produces a hitting set whose size is within $O(\log n)$ of the optimum one. We next prove the constructed hierarchy produces bunches whose sizes are comparable to the randomized construction, and also show that our deterministic construction can be implemented efficiently.

Lemma 6.4.9. *Given an undirected, weighted graph $G = (V, E)$, and a parameter $r \geq 1$, Algorithm 25 computes deterministically, in $O(rmn^{1/r} \log n)$ time, a hierarchy of sets $(A_i)_{0 \leq i \leq r}$ such that for each $v \in V$,*

$$|B(v)| = O(rn^{1/r} \log n).$$

Proof. We start by showing the bound on the size of the bunches. To this end, we first prove by induction on i that $|A_i| \leq n^{1-i/r}$ for all $0 \leq i \leq r - 1$. For the base case, i.e., $i = 0$, the claim is true by construction since $A_0 = V$. We assume that $|A_i| \leq n^{1-i/r}$ for the induction hypothesis, and show that $|A_{i+1}| \leq n^{1-(i+1)/r}$ for the induction step. Note that by construction each set in the collection $\{A_i(v, q, G)\}_{i \in V}$ has size $q = \lceil n^{1/r}(1 + \ln n) \rceil \geq n^{1/r}(1 + \ln n)$. Invoking the greedy algorithm from Lemma 6.4.6, we get a hitting set $A_{i+1} \subseteq A_i$ of size at most

$$\left(\frac{|A_i|}{q} \right) (1 + \ln n) \leq \left(\frac{n^{1-i/r}}{n^{1/r}(1 + \ln n)} \right) (1 + \ln n) = n^{1-(i+1)/r}.$$

We next show that for each $v \in V$ and for each $0 \leq i \leq r - 1$, $|B_i(v)| \leq O(n^{1/p} \log n)$, which in turn implies the claimed bound on the size of vertex

Algorithm 6.5: PREPROCESS$(G, 2r-1)$

```
1 Invoke HierarchyConstruct(G, r), where instead of Steps 1-3 invoke
    DetHierarchy(G, r)
2 for each v ∈ V do
3     Store each B(v), where w ∈ B(v) holds dist_G(v, w).
```

bunches. Note that it suffices to show that $B_i(v) \subseteq A_i(v, q, G)$ since then $|B_i(v)| \leq |A_i(v, q, G)| \leq n^{1/p}(1 + \ln n) = O(n^{1/p} \log n)$. Recall that for $1 \leq i \leq r-1$

$$B_i(v) = \{w \in A_i \setminus A_{i+1} \mid \text{dist}_G(w, v) < \text{dist}_G(A_{i+1}, v)\}$$

Now, by construction of A_{i+1} we have that $A_{i+1} \cap A_i(v, q, G) \neq \emptyset$, which implies that $B_i(v) \subseteq A_i(v, q, G)$ by the definition of $B_i(v)$.

We finally analyze the running time. For $0 \leq i \leq r-2$, consider the sequence of steps in the i-th iteration of the **for** loop in Figure 25. By Lemma 6.4.8, the time to construct the collection of sets $\{A_i(v, q, G)\}_{v \in V}$ is $O(mn^{1/r} \log n)$. Furthermore, since the size of each set in this collection is at least $q = O(n^{1/r} \log n)$, Lemma 6.4.6 guarantees that the greedy algorithm for computing a hitting set A_{i+1} takes $O(n^{1+1/r} \log n)$ time. Combining the above bounds, we get that the total time for the i-th iteration is $O(mn^{1/r} \log n)$. Since there are at most r iterations, we conclude that the running time of the algorithm is $O(rmn^{1/r} \log n)$. □

We now have all the necessary tools to prove Lemma 6.4.4.

Proof of Lemma 6.4.4. We first show how to implement the two operations of the efficient local distance sparsifier $(H, 2r-1)$, and then analyze their running time.

In the preprocessing phase, depicted in Figure 6.5, given the graph G and the stretch parameter $(2r-1)$, we first invoke HierarchyConstruct(G, r) in Figure 6.3, where Steps 1-3 are replaced by the deterministic algorithm for computing the hierarchy of sets DetHierarchy(G, r). Note that this modification ensures that our preprocessing algorithm is deterministic. Next, for each vertex $v \in V$, we store its bunch $B(v)$ in a balanced binary search tree, where each vertex $w \in B(v)$ has as key the value $\text{dist}_G(w, v)$ (this step could be implemented differently, but as we will shortly see, it will be useful in the subsequent applications of our algorithm).

We next describe how to implement the query operation, depicted in Figure 6.6. Let K be the set of queried terminals. The main idea to construct a vertex distance sparsifier $H[K]$ of G with respect to K is to exploit the bunches that we stored in the preprocessing step. More concretely, let $H[K]$ be an initially empty graph. For each vertex $v \in K$, and every vertex in its bunch $u \in B(v)$, we add to $H[K]$ the edge (u, v) with weight $\text{dist}_G(u, v)$. To show that the resulting graph $H[K]$ is indeed a vertex distance sparsifier with respect to K, we briefly review the query algorithm in the construction of Thorup and Zwick [251], and show that this immediately applies to our graph setting.

Algorithm 6.6: QUERYSPARSIFIER(G, K)

```
Set H[K] ← ∅
for each v ∈ K do
    for every u ∈ B(v) do
        Add (v, u) to E(H[K]) with weight dist_G(v, u)
return H[K]
```

Let $u, v \in K$ by any two terminals. The algorithm uses the variables w and i, and starts by setting $w \leftarrow u$, and $i \leftarrow 0$. Then it repeatedly increments the value of i, swaps u and v, and sets $w \leftarrow p_i(u) \in B(u)$, until $w \in B(v)$. Finally, it returns a distance estimate $\delta(u, v) = \text{dist}_G(w, u) + \text{dist}_G(w, v)$. Observe that $w = p_i(u) \in B(u)$ for some $0 \leq i \leq r - 1$ and $w \in B(v)$. By construction of our vertex sparsifier $H[K]$, note that the edges (w, u) and (w, v), and their corresponding weights, $\text{dist}_G(w, u)$ and $\text{dist}_G(w, v)$, are added to $H[K]$. Thus, there must exist a path between u and v in $H[K]$ whose stretch is at most the stretch of the distance estimate $\delta(u, v)$. Since in [251] it was shown that for every $u, v \in K$,

$$\text{dist}_G(u, v) \leq \delta(u, v) \leq (2r - 1)\text{dist}_G(u, v),$$

we immediately get that

$$\text{dist}_G(u, v) \leq \text{dist}_{H[K]}(u, v) \leq (2r - 1)\text{dist}_G(u, v).$$

We finally analyze the running time for both operations. First, note that by Lemma 6.4.9, the deterministic algorithm for constructing the hierarchy of sets DETHIERARHCY(G, r) runs in $O(rmn^{1/r} \log n)$ time. Moreover, Thorup and Zwick [251] showed that given a hierarchy of sets, the bunches for all vertices in G can be computed in $O(rmn^{1/r} \log n)$ time. Combining these two bounds we get that the operation PREPROCESS(G, r) runs in $O(rmn^{1/r} \log n) = \tilde{O}(mn^{1/r})$ time. For the running time of QUERYSPARSIFIER(G, K), recall that $H[K]$ consists of the union over all bunches of terminal vertices in K. Since the size of a each individual vertex bunch is bounded by $O(rn^{1/r} \log n)$ (Lemma 6.4.9), we get that the size of $H[K]$ is bounded by $O(|K|rn^{1/r} \log n) = \tilde{O}(|K|n^{1/r})$. The latter also bounds the time to output $H[K]$. □

We next show that local sparsifiers for distances are closed under transitivity and decomposition. While transitivity follows directly from the definition, for the sake of completeness we include the proof for decomposability.

Lemma 6.4.10 (Transitivity). *If H_1 is an α_1-local sparsifier of G, and H_2 is a α_2-beta local sparsifier of H_1, then H_2 is an $\alpha_1\alpha_2$-local sparsifier of G.*

Lemma 6.4.11 (Decomposability). *Let $G = (V, E)$ be an undirected, weighted graph, let E_1, E_2 be a partition of the edge set E, and let H_i be an α_i-local sparsifier of $G_i = (V, E_i)$, for each $1 \leq i \leq 2$. Then $H = H_1 \cup H_2$ is a $\max\{\alpha_1, \alpha_2\}$-local sparsifier of H.*

Proof. Let $u, v \in V$ be an arbitrary pair of vertices, and let $P_G(u,v)$ be the shortest path distance of weight $\text{dist}_G(u,v)$ between u and v in G. Moreover, for each $1 \leq i \leq 2$ let $P_{G_i}(u,v)$ be the edges of $P_G(u,v)$ that belong to E_i. Since H_i is an α_i-sparsifier of G_i, for each $1 \leq i \leq r$, we know that for each edge $e \in P_{G_i}(u,v)$, there exists a path $P_{H_i}(e)$ in H_i such that

$$\mathbf{w}(e) \leq \mathbf{w}(P_{H_i}(e)) \leq \alpha_i \cdot \mathbf{w}(e). \tag{6.5}$$

Define

$$\mathbf{w}(P_H(u,v)) := \sum_{1 \leq i \leq 2} \sum_{e \in P_{G_i}(u,v)} \mathbf{w}(P_{H_i}(e)),$$

to be a path between u and v in H. We next show that weight of this path dominates as well as stretches $\text{dist}_G(u,v)$ within a $\max\{\alpha_1, \alpha_2\}$ factor.

Indeed, repeatedly applying Equation 6.5 we have that

$$\begin{aligned}\mathbf{w}(P_H(u,v)) &= \sum_{1 \leq i \leq 2} \sum_{e \in P_{G_i}(u,v)} \mathbf{w}(P_{H_i}(e)) \geq \sum_{1 \leq i \leq 2} \sum_{e \in P_{G_i}(u,v)} \mathbf{w}(e) \\ &= \mathbf{w}(P_G(u,v)) = \text{dist}_G(u,v),\end{aligned}$$

and,

$$\begin{aligned}\mathbf{w}(P_H(u,v)) &= \sum_{1 \leq i \leq 2} \sum_{e \in P_{G_i}(u,v)} \mathbf{w}(P_{H_i}(e)) \leq \max\{\alpha_1, \alpha_2\} \cdot \sum_{1 \leq i \leq 2} \sum_{e \in P_{G_i}(u,v)} \mathbf{w}(e) \\ &= \max\{\alpha_1, \alpha_2\} \cdot \mathbf{w}(P_G(u,v)) = \max\{\alpha_1, \alpha_2\} \cdot \text{dist}_G(u,v).\end{aligned}$$

□

We now have all the necessary tools to prove Theorem 6.4.1.

Proof of Theorem 6.4.1. Let $(H, 2r-1, \tilde{O}(n^{1/r}), \tilde{O}(n^{1/r}))$ be an efficient distance local sparsifier of G (Lemma 6.4.4), which is closed under transitivity and decomposition (Lemmas 6.4.10 and 6.4.11). Plugging the parameters $\alpha = (2r-1)$, $f(n) = \tilde{O}(n^{1/r})$, $g(n) = \tilde{O}(n^{1/r})$, $h(n) = 1$ into Theorem 6.3.1 we get an incremental algorithm such that for any pair of vertices u and v it reports a query estimate $\delta(u,v)$ with

$$\text{dist}_G(u,v) \leq \delta(u,v) \leq (2r-1)^{\ell} \text{dist}_G(u,v),$$

and handles update and query operations in worst-case time of

$$\tilde{O}\left(\left(\sum_{j=1}^{\ell} \frac{\beta_{j-1}}{\beta_j} + \beta_\ell\right) n^{2/r}\right), \quad \text{where } \beta_0 = m.$$

Note that the choice of parameters $\{\beta\}_{0 \leq i \leq \ell}$ does not depend on the factor $n^{2/r}$. Therefore, by ignoring this and applying Lemma 6.3.6 we get that there exists a choice of parameters $\{\beta\}_{0 \leq i \leq \ell}$ such that

$$\tilde{O}\left(\left(\sum_{j=1}^{\ell} \frac{\beta_{j-1}}{\beta_j} + \beta_\ell\right) n^{2/r}\right) = \tilde{O}\left(m^{1/(\ell+1)} n^{2/r}\right) = \tilde{O}(n^{2/(\ell+1)} n^{2/r}).$$

□

6.5 Incremental All Pair Max-Flow

In this section we show how to use our general Theorem 6.3.1 to design online incremental algorithms for the approximate All-Pair Max-Flow Problem with fast worst-case update and query time. Concretely, we will show that that assumptions (1) and (2) in Theorem 6.3.1 are satisfied with certain parameters for flows. Note that (3) follows immediately by employing the $\tilde{O}(m)$ time approximate (s,t)-maximum flow algorithm due to Peng [210]. We have the following theorem.

Theorem 6.5.1. *Let $G = (V, E)$ be an undirected, weighted graph. For every $\ell \geq 1$, there is an incremental (randomized) approximate All Pair Max Flow algorithm that maintains for every pair of nodes u and v, a maximum flow estimate $\delta(u, v)$ such that*

$$\frac{1}{O(\log^{4\ell} n)} \text{max-flow}_G(u, v) \leq \delta(u, v) \leq \text{max-flow}_G(u, v),$$

with wort-case update and query time of

$$\tilde{O}(n^{2/(\ell+1)}).$$

We start by introducing the usual definitions of sparsifiers and vertex sparsifiers for flows. Having defined these, the definition of local sparsifiers becomes apparent from the general definition we introduced in Section 6.2. Let $G = (V, E)$ be a undirected, weighted graph with a *terminal* set $K \subseteq V$. Let $\mathbf{d}$ be a demand function over K in G such that $\mathbf{d}(x, x') = \mathbf{d}(x', x)$ and $\mathbf{d}(x, x) = 0$ for all $x, x' \in K$. We denote by $P_{x,x'}$ the set of all paths between x and x' in G, for all $x, x' \in K$. Further, for each edge $e \in E$, let P_e be the set of all paths using edge e. A *concurrent* (*multi-commodity*) flow $\mathbf{f}$ of *congestion* λ is function over terminal paths in G such that (1) $\sum_{p \in P_{x,x'}} \mathbf{f}(p) \geq \mathbf{d}(x, x')$, for all distinct terminal pairs $x, x' \in K$, and (2) $\sum_{p \in P_e} \mathbf{f}(p) \leq \lambda c(e)$, for all $e \in E$. We let $\text{cong}_G(\mathbf{d})$ denote the congestion of the concurrent flow that attains the smallest congestion.

Definition 6.5.2 (Sparsifiers for Flow)**.** *Let $G = (V, E)$ be an undirected, weighted graph. A graph $H = (V', E')$ with $V \subseteq V'$ is a flow sparsifier of G with quality $\alpha \geq 1$ iff for every demand function $\mathbf{d}$ among any pair of vertices in V*

$$\text{cong}_H(\mathbf{d}) \leq \text{cong}_G(\mathbf{d}) \leq \alpha \cdot \text{cong}_H(\mathbf{d}).$$

Definition 6.5.3 (Vertex Sparsifiers for Flows)**.** *Let $G = (V, E)$ be an undirected, weighted graph with a terminal set $K \subset V$. A graph $H = (V, E')$ with $K \subseteq V'$ is a (vertex) flow sparsifier of G with quality $\alpha \geq 1$ iff for every demand function $\mathbf{d}$ among any pair of vertices in K*

$$\text{cong}_H(\mathbf{d}) \leq \text{cong}_G(\mathbf{d}) \leq \alpha \cdot \text{cong}_H(\mathbf{d}).$$

We next show that the flow property in graphs admits efficient local sparsifier with desirable guarantees.

Lemma 6.5.4 (Efficient Flow Local Sparsifiers)**.** *Given an undirected, weighted graph $G = (V, E)$, there is a randomized algorithm that constructs an efficient flow local sparsifier with $O(\log^4 n)$ quality, $\tilde{O}(m)$ preprocessing time, and $\tilde{O}(|K|)$ query time, where K is any set of queried terminals.*

We prove the above lemma by using and slightly extending the fast cut-based decomposition tree due to Räcke, Shah and Täubig [216], and Peng [210]. We remark that their result is stated only for unweighted graphs, but it easily extends to the weighted case.

Theorem 6.5.5 ([210, 216])**.** *Given an undirected, weighted graph $G = (V, E)$, there is an $\tilde{O}(m)$ time randomized algorithm FlowSparsify(G) that with high probability computes a flow sparsifier $H = (V', E')$ with $V \subseteq V'$ satisfying the following properties*

1. *H is a bounded degree rooted tree*
2. *H has quality $O(\log^4 n)$*
3. *The leaf nodes of H correspond to nodes in G,*
4. *The height of H is at most $O(\log^2 n)$.*

Proof. The original construction of Räcke et al. [216] produces a rooted tree H' which satisifes the above properties, except that H' has unbounded degree and the height of the tree is $O(\log n)$. Since we will exploit the bounded degree assumption in the subsequent applications of our data-structure, here we present a standard reduction from H' to a bounded degree H at the cost of increasing the height of the tree by a logarithmic factor.

Let H' be the rooted tree we described above. Let $u \in H'$ be an internal node of degree larger than 2 and let $C(u)$ be its children. We start by removing all edges incident to the children $C(u)$ from H', and record all their corresponding edge weights. Next, we create a bounded degree rooted tree $\tilde{H}$ where the children $C(u)$ are the leaf nodes, i.e., $L(\tilde{H}) = C(u)$, and u is the root of $\tilde{H}$. To complete the construction of $\tilde{H}$ we need to define its edge weights. To this end, for any subtree $R \subseteq \tilde{H}$ let $E(L(R))$ denote the set of edges incident to leaf nodes in R. We distinguish the following two cases. (1) If $e = (x, y) \in E(L(\tilde{H}))$ and $x \in L(\tilde{H}) = C(u)$, we set $\mathbf{w}_{\tilde{H}}(x, y) = \mathbf{w}_{H'}(x, u)$. (2) If $e = (x, y) \notin E(L(\tilde{H}))$, then let $\tilde{H}_x$ and $\tilde{H}_y$ be the trees obtained after deleting the edge e from $\tilde{H}$. Further, for any subtree $R \subseteq \tilde{H}$ define

$$\mathbf{w}(R) := \sum_{e \in E(L(R))} \mathbf{w}_{\tilde{H}}(e).$$

Finally, for $e = (x, y) \notin E(L(\tilde{H}))$ and $e \in \tilde{H}$ we set

$$\mathbf{w}_{\tilde{H}}(x, y) = \min\{\mathbf{w}(\tilde{H}_x), \mathbf{w}(\tilde{H}_y)\}.$$

Note that the weight sums $\mathbf{w}(\tilde{H}_x)$ and $\mathbf{w}(\tilde{H}_x)$ can be calculated since we first defined the weights for edges in $E(L(\tilde{H}))$. Also observe that H' remains a tree because we simply removed children of u (which could be viewed as a star) and replaced this by another bounded degree tree $\tilde{H}$. We repeat the above process for every internal node of H' until H' becomes a bounded degree rooted tree, and denote by H the final resulting tree.

We claim that H has depth at most $O(\log^2 n)$. Recall that the initial height of H' was $O(\log n)$, and every replacement of the star centered at a non-terminal with a bounded degree tree increases the height by an additive of $O(\log n)$. Summing up over $O(\log n)$ levels, we get the claimed bound.

Finally, it is easy to see that H is flow sparsifier of quality 1 for H' with respect to all leaf nodes of H', which in turn correspond to the nodes of graph G. Thus, H is also a flow sparsifier for G with quality $O(\log^4 n)$. □

We now have all the necessary tools to prove Lemma 6.5.4.

Proof of Lemma 6.5.4. We show how to implement the two operations of the efficient local flow sparsifier $(H, O(\log^4 n))$, argue about its correctness, and then analyze the running time of each operation.

In the preprocessing phase, given a graph G, we simply invoke FlowSparsify(G) from Theorem 6.5.5 and let H be the resulting sparsifier. For implementing the query operation, let K denote the set of queried terminals. The main idea for constructing a (vertex) flow sparsifier $H[K]$ of G with respect to K is to exploit the fact that H is a tree. Concretely, let $H[K]$ be an initially empty graph. For $v \in K$, let $P(v, r, H)$ be the path between v and r in H, where r is the root of H (since $v \in K \subseteq V$, recall that v is a leaf node of H by Property (3) in Theorem 6.5.5). For each $v \in K$, and every edge $e \in P(v, r, H)$, we add e with weight $w_H(e)$ to $H[K]$. Finally, we return $H[K]$ as a (vertex) flow sparsifier of G with respect to K.

We now argue about the correctness of $H[K]$. First, we show that $H[K]$ is a quality 1 (vertex) flow sparsifier of H with respect to K. To see this, note that since H is a tree, every (multi-commodity) flow among any two leaf vertices (u, v) is routed according to the unique shortest path between between u and v in H, denoted by $P(u, v, H)$. Since $H[K]$ is formed taking the union of the paths $P(v, r, H)$, for each $v \in K$, and $P(u, v, H) \subseteq (P(v, r, H) \cup P(u, r, H))$, it follows that $P(u, v, H)$ is also contained in $H[K]$. Thus every flow we can route in H among any two pairs in K, we can feasible route in $H[K]$. For the next direction, observe that by construction $H[K] \subseteq K$. Therefore, any flow among any two pairs in K that can be feasibly routed in $H[K]$, can also be routed in H (this follows since H has more edges than $H[K]$, and thus the routing in H has more flexibility). Combining the above we get that $H[K]$ is a quality 1 (vertex) flow sparsifier of H. Since H is flow sparsifier of G with quality $O(\log^4 n)$ (Property (2) in Theorem 6.5.5) and $K \subseteq V$, applying transitivity on $H[K]$ and H (which we will shortly prove) we get that $H[K]$ is a quality $O(\log^4 n)$ (vertex) flow sparsifier of G with respect to K.

We finally analyze the running time for both operations. Recall that the operation PREPROCESS(G) is implemented by simply invoking FLOWSPARSIFIY(G). By Theorem 6.5.5, we know that the latter can be implemented in $\tilde{O}(m)$, which in turn bounds the running time of our preprocessing step. For the running time of QUERYSPARSIFIER(G, K), recall that $H[K]$ consists of the union over the paths $P(v, r, H)$, for each $v \in K$. Since the length of each such path is bounded by $O(\log^2 n)$ (Property (4) in Theorem 6.5.5), we get that the size of $H[K]$ is bounded by $O(|K| \log^2 n) = \tilde{O}(|K|)$. Note that after having access to any leaf vertex v, the path $P(v, r, H)$ can be retrieved from H in time proportional to its length. This implies that the time to output $H[K]$ is also bounded by $\tilde{O}(|K|)$. □

We next show that local sparsifiers for flows are closed under transitivity and decomposition. While transitivity follows directly from the definition, for the sake of completeness we include the proof for decomposability.

Lemma 6.5.6 (Transitivity). *If H_1 is an α_1-local sparsifier of G, and H_2 is an α_2-local sparsifier of H_1, then H_2 is an $\alpha_1\alpha_2$-local sparsifier of G.*

Lemma 6.5.7 (Decomposability). *Let $G = (V, E)$ be an undirected, weighted graph, let E_1, E_2 be a partition of the edge set E, and let H_i be an α_i-local sparsifier of $G_i = (V, E_i)$, for each $1 \leq i \leq 2$. Then $H = H_1 \cup H_2$ is an $\max\{\alpha_1, \alpha_2\}$-local sparsifier of H.*

Proof. Consider a demand $\mathbf{d}$ among any pair of vertices $u, v \in V$ that is routable in G. Let $\mathbf{f}$ a (multi-commodity) flow that routes $\mathbf{d}$, and let $D = \{(p_1, \mathbf{f}(p_1)), (p_2, \mathbf{f}(p_2)), \ldots, (p_\ell, \mathbf{f}(p_\ell))\}$ be a flow-decomposition, where p_i is a path, and $\mathbf{f}(p_i)$ is the amount of flow set along this path. Note that a flow path decomposition also specifies a demand since for any $u, v \in V$, $\mathbf{d}(u, v) = \sum_{p \in D(u,v)} \mathbf{f}(p)$, where $D(u, v)$ is all the paths in D whose endpoints are exactly u and v. Fix any path $p \in D$, and let $p^{(1)}$ and $p^{(2)}$ be the set of subpaths of p that use only edges from G_1 and G_2, respectively (note that $p^{(1)}$ and $p^{(2)}$ partition p). Note that the set of paths $p^{(1)}$ and $p^{(2)}$ induce demands in G_1 and G_2. Taking the union over all paths $p \in D$ will induce demands $\mathbf{d}_1$ in G_1 and $\mathbf{d}_2$ in G_2 with $\mathbf{d} = \mathbf{d}_1 + \mathbf{d}_2$, and these demands are routed among flow paths that lie entirely within G_1 or G_2. By the definition of flow-sparsifier, these demands are also routable in H_1 and H_2, and hence the demand $\mathbf{d}_1 + \mathbf{d}_2 = \mathbf{d}$ is routable in H.

For the other direction assume that a demand $\mathbf{d}$ among any pair of vertices $u, v \in V$ is routable in H. Similarly to above, let D be the corresponding path decomposition of the flow $\mathbf{f}$ that routes $\mathbf{d}$. Fix any path $p \in D$, and let $p^{(1)}$ and $p^{(2)}$ be the set of subpaths of p that use only edges from H_1 and H_2. Note that H_1 and H_2 might have extra vertices that do not belong to V. However, the endpoints of every path p' belonging to $p^{(1)} \cup p^{(2)}$ must be from V. This means that the these paths induced in H_1, and H_2 are among pairs of vertices in V. Thus, taking the union over all paths $p \in D$ will induce demands $\mathbf{d}_1$ in H_1 and $\mathbf{d}_2$ in H_2 with $\mathbf{d} = \mathbf{d}_1 + \mathbf{d}_2$, which are routed among flows path that lie entirely in H_1 and H_2,

respectively. By the definition of flow sparsifiers, these demands routed in G_1 and G_2 with congestion $\max\{\alpha_1, \alpha_2\}$, respectively. Thus we can also route their sum $\mathbf{d} = \mathbf{d}_1 + \mathbf{d}_2$ with congestion $\max\{\alpha_1, \alpha_2\}$ in G. □

We now have all the necessary tools to prove Theorem 6.5.1.

Proof of Theorem 6.5.1. Let $(H, O(\log^4 n), \tilde{O}(1), \tilde{O}(1))$ be an efficient flow local sparsifier of G (Lemma 6.5.4), which is closed under transitivity and decomposition (Lemmas 6.5.6 and 6.5.7). Plugging the parameters $\alpha = O(\log^4 n)$, $f(n) = \tilde{O}(1)$, and $g(n) = \tilde{O}(1)$ in Theorem 6.3.1 we get an incremental algorithm such that for any pair of vertices u and v it reports a query estimate $\delta(u, v)$ with

$$\frac{1}{\tilde{O}(\log^{4\ell} n)} \text{max-flow}_G(u, v) \leq \delta(u, v) \leq \text{max-flow}_G(u, v),$$

and handles update and query operations in worst-case time of

$$\tilde{O}\left(\left(\sum_{j=1}^{\ell} \frac{\beta_{j-1}}{\beta_j} + \beta_\ell\right)\right), \quad \text{where } \beta_0 = m.$$

Note that the choice of parameters $\{\beta\}_{0 \leq i \leq \ell}$ does not depend on the factor $\text{poly}(\log n))^2$. Therefore, by ignoring this and applying Lemma 6.3.6 we get that there exists a choice of parameters $\{\beta\}_{0 \leq i \leq \ell}$ such that

$$\tilde{O}\left(\left(\sum_{j=1}^{\ell} \frac{\beta_{j-1}}{\beta_j} + \beta_\ell\right)\right) = \tilde{O}\left(m^{1/(\ell+1)}\right) = \tilde{O}(n^{2/(\ell+1)}).$$

□

6.6 Incremental Tree Flow Sparsifier (Räcke Tree)

In this section we show that a slightly modified version of the algorithm used to prove Theorem 6.3.1 and a few extensions allow us to design a fast incremental algorithm for maintaing a (multi-commodity) flow sparsifier H of a graph G with poly-logarithmic quality. Most importantly H will be a tree graph that satisfies certain interesting properties that we will exploit to maintain other dynamic problems.

Our extensions build upon the following two main ideas. First, we want to argue that the efficient local sparsifier is a tree. Indeed, observe that the efficient local sparsifier H produced by Lemma 6.5.4 produces a tree (Property (1)), and moreover, by definition of local sparsifiers, the vertex sparsifier $H[K]$ that we query from H with respect to any set of terminals K must also be a tree. Throughout we will refer to H as a *tree flow sparsifier*. Now, recall that in Algorithm 6.1 we have an update rule for rebuilding tree flow sparsifiers. Our goal is to show that under this update rule, the updated sparsifiers still remain trees. We observe that this becomes clear once one formalizes the update process, as shown below.

Let H be a tree flow sparsifier of $G = (V, E)$, let E_0 be some set of edges with $V(E_0) \subseteq V$, and let $H[V(E_0)]$ be a (vertex) flow sparsifier of G obtained by querying H with respect to $V(E_0)$. Moreover, let $H'[V(E_0)]$ be a tree flow sparsifier of $H[V(E_0)] \cup E_0$. Then we have that

$$H' := (H \setminus H[V(E_0)]) \cup H'[V(E_0)]$$

is indeed a tree flow sparsifier of $G \cup E_0$.

The second idea we need is to ensure that at any point of time our incremental algorithm maintains a tree flow sparsifier. Note that this is not the case in Algorithm 6.1 since for answering queries (see Algorithm 6.2) it was sufficient to consider the sparsifier $H_{\ell-1}$ plus the edge set E_ℓ. To overcome this, we simply maintain an additional tree flow sparsifier H_ℓ at level ℓ of the hierarchy, and after each edge insertion we rebuild H_ℓ. Concretely, H_ℓ is updated by the above rule using the sparsifier $H_{\ell-1}$, the edge set E_ℓ and the (vertex) flow sparsifier $H_{\ell-1}[V(E_\ell)]$ that is obtained by querying $H_{\ell-1}$ with respect to $V(E_\ell)$. This modification gives that H_ℓ is tree flow sparsifier of G at any point of time at the cost of increasing the quality guarantee by a poly-logarithmic factor but not affecting our running time guarantee.

Combining the above ideas leads to the following theorem.

Theorem 6.6.1. *Let $G = (V, E)$ be an undirected, weighted graph. For every $\ell \geq 1$, there is an incremental (randomized) algorithm that maintains a tree flow sparsifier H of G with quality $O(\log^{8\ell} n)$ and depth $O(\ell \log^2 n)$. The worst-case update time is $\tilde{O}(n^{2/(\ell+1)})$.*

6.6.1 Applications of tree flow sparsifiers

We next show how to apply Theorem 6.6.1 for designing efficient incremental algorithm for cut/flow based problems.

Incremental Maximum Concurrent Flow. Recall from Section 6.5 that $\mathrm{cong}_G(\mathbf{d})$ is the congestion of the concurrent flow that attains the smallest congestion among all flows that route demand $\mathbf{d}$ supported on the terminals K. Recall from Section 6.1 that given k demand pairs $\{(s_i, t_i, \mathbf{d}(i))\}_{i=1}^k$, the vector $\mathbf{d}$ will have $O(k)$ non-zero entries. In the *Maximum Concurrent Flow Problem* the we want to find a flow that minimizes $\mathrm{cong}_G(\mathbf{d})$.

The fastest approximation algorithm for solving the Maximum Concurrent Flow Problem is due to Sherman [229].

Theorem 6.6.2 ([229])**.** *Let $\varepsilon > 0$. Given an undirected, weighted graph $G = (V, E)$ and a demand vector $\mathbf{d}$ describing k demand pairs, there is an $\tilde{O}(mk)$ algorithm that approximates $\mathrm{cong}_G(\mathbf{d})$ within a $(1 + \varepsilon)$ factor.*

In the dynamic version of this problem, we want to construct a data-structure that supports the following operations

- INSERT(u, v): insert the edge (u, v) in the graph, and
- QUERY$(\mathbf{d})$: return the congestion $\mathrm{cong}_G(\mathbf{d})$ for routing demand $\mathbf{d}$ in the current graph G.

Now, given Theorem 6.6.1, we just maintain tree flow sparsifier H. Then, given a query $\{(s_i, t_i, \mathbf{d}_i)\}_{i=1}^k$ describing k demand pairs, we do the following. Let K be the terminals including all s_i and t_i. Then, we just run Sherman's algorithm on $H[K]$ (which is the union of root-to-leaf paths of all nodes in K). This leads to the following corollary.

Corollary 6.6.3. *For every $\ell \geq 1$, there is an incremental (randomized) approximate Maximum Concurrent Flow algorithm that maintains for every demand $\mathbf{d}$ describing k demand pairs an estimate $\delta(\mathbf{d})$ such that*

$$\mathrm{cong}_G(\mathbf{d}) \leq \delta(\mathbf{d}) \leq O(\log^{8\ell} n)\,\mathrm{cong}_G(\mathbf{d}),$$

with wort-case update of $\tilde{O}(n^{2/(\ell+1)})$ and query time of $\tilde{O}(k^2)$).

Uniform sparsest cut and cut oracles. Recall that the uniform sparsest cut Φ_G of a weighted graph G is defined as

$$\Phi_G = \min_{\emptyset \neq S \subset V} \frac{\mathrm{cap}_G(S, V \setminus S)}{|S| \cdot |V \setminus S|}$$

where $\mathrm{cap}_G(S, V \setminus S) = \sum_{(u,v)\in E, u\in S, v\notin S} \mathbf{w}(u, v)$.

In the *dynamic uniform sparsest cut* problem, we want to approximate Φ_G when given a query. In the *dynamic cut oracle* problem, we want to maintain a data structure such that, given a set of nodes S, we can approximate $\mathrm{cap}_G(S, V \setminus S)$ in time proportional to $|S|$.

By Theorem 6.6.1, the above problems reduce to solving them on a tree that undergoes changes. More importantly, this tree has only polylogarithmic depth. Employing standard techniques for maintaining information on a dynamic tree (e.g. ET tree [129] link/cut tree [232] or top tree [19]) leads to the following corollaries.

Corollary 6.6.4. *For every $\ell \geq 1$, there is an incremental (randomized) $O(\log^{8\ell} n)$-approximate uniform sparsest cut algorithm with wort-case update of $\tilde{O}(n^{2/(\ell+1)})$. Given a query, the algorithm returns a $O(\log^{8\ell} n)$-approximation to the uniform sparsest cut in $O(1)$ time.*

Corollary 6.6.5. *For every $\ell \geq 1$, there is an incremental (randomized) cut oracle algorithm with wort-case update of $\tilde{O}(n^{2/(\ell+1)})$. Given a set S of nodes, the algorithm returns an $O(\log^{8\ell} n)$-approximation to the size of the cut induced by S, i.e. $\mathrm{cap}_G(S, V \setminus S)$, in time $\tilde{O}(|S|)$.*

6.7 From Vertex Sparsifiers to Offline Dynamic Algorithms

In this section we show how to use efficient vertex sparsifier constructions to design *offline* (approximate) dynamic algorithms for graph problems with certain properties while achieving fast amortized update and query time. To achieve this we use a framework that has been exploited for solving offline 3-connectivity [211]. Our main contribution is to show that this generalizes to a much wider class of problems, leading to several interesting bounds which are not yet known in the *online* dynamic graph literature.

We start by defining the model. We are given an undirected graph $G = (V, E)$ and an *offline* sequence of events or operations $x_1, \ldots, x_m$, where x_i is ether an edge update (insertion or deletion), or a query q_i which asks about some graph property in G at time i. The goal is to process this sequence of updates in G while spending total time proportional to $O(mf(m))$, where $f(m)$ is ideally some sublinear function in m.

We next show that an analogue to Theorem 6.3.1 can also be obtained in the offline graph setting. Our algorithm makes use of the notion of vertex sparsifiers as well as their useful properties including transitivity and decomposability (see Section 6.2). In our construction we want graph properties that admit (1) fast algorithms for computing vertex sparsifiers and (2) guarantee that the size of such sparsifers is reasonably small. We formalize these requirements in the following definition.

Definition 6.7.1. *Let $G = (V, E)$ be a graph, with a terminal set $K \subseteq V$ and let $f(n), g(n) \geq 1$ be functions. We say that $(G', \alpha, f(n), g(n))$ is an α-efficient vertex sparsifier of G with respect to K iff G' is an α-vertex sparsifier of G, the time to construct G' is $O(m \cdot f(n))$, and the size of G' is $O(|K| \cdot g(n))$.*

Theorem 6.7.2. *Let $G = (V, E)$ be a graph, and for any $u, v \in V$, let $\mathcal{P}(u, v, G)$ be a solution to a minimization problem between u and v in G. Let $f(n), g(n), h(n) \geq 1$ be functions, $\alpha, \ell \geq 1$ be parameters associated with the approximation factor, and let $\beta_0, \beta_1, \ldots, \beta_\ell$ with $\beta_0 = m$ be parameters associated with the running time. Assume the following properties are satisfied*

1. *G admits an efficient vertex sparsifier $(G', \alpha, f(n), g(n))$,*
2. *G' is transitive and decomposable,*
3. *The property $\mathcal{P}(u, v, G)$ can be computed in $O(mh(n))$ time in a graph with m edges and n vertices.*

Then there is an offline (approximate) dynamic algorithm that maintains for every pair of nodes u and v, an estimate $\delta(u, v)$, such that

$$\mathcal{P}(u, v, G) \leq \delta(u, v) \leq \alpha^\ell \cdot \mathcal{P}(u, v, G). \tag{6.6}$$

The total time for processing a sequence of m operations is:

$$\tilde{O}\left(\beta_0\left(\sum_{j=1}^{\ell}\left(\frac{\beta_{j-1}}{\beta_j}\right)f(n)+\beta_\ell h(n)\right)g(n)\right) \quad \textit{where } \beta_0 = m. \tag{6.7}$$

Before describing the underlying data-structure upon which the above theorem builds, we reduce the arbitrary sequence of operations into a more structured one, and also build a particular view for the problem. These will allow us to greatly simplify the presentation.

Concretely, first we may assume that each edge is inserted and deleted exactly once during the sequence of operations. We achieve this by simply treating each edge instance as a new edge, i.e., we assume that each insertion of an edge $e = (u, v)$ inserts a new edge that is different from all previous instances of (u, v).

Second, since we are given the entire sequence of operations, for each edge e we associate an interval $[i_e, d_e]$ which indicates the insertion and deletion time of e in the operation sequence. Furthermore, we denote by q_t the time when query q was asked in the operation sequence. Let $[1, m]$ denote the interval covering the entire event sequence. If we are interested in processing updates from a given interval $[r, s]$, we will define graphs that consists of two types of edges with respect to this interval:

1. *non-permanent edges*, which are edges affected by an event in this interval, i.e., $E^p_{[r,s]} = \{e \mid i_e \text{ or } d_e \in [r, s]\}$,
2. *permanant edges*, which are edges present throughout the entire interval, i.e., $E^{np}_{[r,s]} = \{e \mid i_e < r \leq s < d_e\}$.

Additionally, it will be useful to define the queried vertex pairs within the interval $[r, s]$: $Q_{[r,s]} = \{q \mid q_t \in [r, s]\}$.

Data Structure. We now describe a generic tree data-structure T, which allows us to unify our framework and thus greatly simplify the presentation. This tree structure is obtained by hierarchically partitioning the operation sequence into smaller disjoint intervals. These intervals induce graphs that are suitable for applying vertex sparsifiers, which in turn allow us to process updates in a fast way, while paying some error in the accuracy of the query operations.

Consider some integer parameter $\ell \geq 1$ and parameters $\beta_0, \beta_1, \ldots, \beta_\ell$ with $\beta_0 = m$. The tree T has $\ell + 1$ levels, where each level i is associated with the parameter $\beta_i, i = 0, \ldots, \ell$. Each node of the tree stores some interval from the event sequence. Formally, our decomposition tree T satisfies the following properties:

1. The root of the tree stores the interval $[1, m]$.
2. The intervals stored at nodes of same level are disjoint.
3. Each interval $[r, s]$ stored at a node in T is associated with

- a graph $G_{[r,s]} = \left(V, E^p_{[r,s]}\right)$,
- a graph of *new permanent edges* $H_{[r,s]} = G_{[r,s]} \setminus G_{[q,t]}$, where $G_{q,t}$ is the parent of $G_{[r,s]}$ in T (if any).
- a set of *boundary vertices* $\partial_{[r,s]} = V(E^{np}_{[r,s]}) \cup V(Q_{[r,s]})$.

4. If $[r,s] \subseteq [q,t]$ then it holds that (a) $\partial_{[r,s]} \subseteq \partial_{[q,t]}$, and (b) $E^p_{[q,t]} \subseteq E^p_{[r,s]}$.
5. The length of the interval stored at a node at level i is β_i.
6. A node at level i has β_i/β_{i+1} children.
7. The number of nodes at level i is at most $O(\beta_0/\beta_i)$.

The lemma below shows that a decomposition tree can be constructed in time proportional to the length of the operation sequence times the height of the tree.

Lemma 6.7.3. *Let $G = (V, E)$ be a dynamic graph where the sequence of operations is revealed upfront. Then there is an algorithm that computes the decomposition tree T in $O(\ell m)$ time, where m denotes the length of the operation sequence and ℓ is the height of the tree.*

Proof. Let T be a tree with a single node (corresponding to its root) that stores the interval $[1, m]$. We augment T in the following natural way: (a) We partition the interval $[1, m]$ into $\beta_0/\beta_1 = m/\beta_1$ disjoint intervals, each of length β_1. (b) For each of these intervals we create a node in the tree T, and connect each node with the root of T, i.e., those nodes form the children of the root, and thus the nodes at level 1 of T. (c) We recursively apply steps (a) and (b) to the newly generated nodes until we reach the $(\ell + 1)$-st level of the tree.

By the construction above, it easily follows that the generated tree T satisfies properties (1), (2), (4), (5), (6) and (7). Thus, it remains to show how to compute the quantities in (3). This can be achieved by (a) computing the intervals $[i_e, d_e]$, for every edge e in the sequence (note that this is possible because we assumed that every edge is inserted and deleted exactly once within the interval $[1, m]$), and (2) for each node in the tree, computing the sets $E^{np}_{[r,s]}$ and $E^p_{[r,s]}$.

For the running time, observe that computing the intervals $[i_e, d_e]$ takes $O(m)$ time. Having computed these intervals, we can level-wise compute the permanent and non-permanent edges for each node in that particular level. By disjointedness of the intervals, the amount of work we perform per level is $O(m)$. Since there are most $O(\ell)$ levels, it follows that the running time for constructing the decomposition tree is $O(\ell m)$. □

Computing vertex sparsifiers in the hierarhcy. We next show how to efficiently compute a vertex sparsifier $G'_{[r,s]}$ for each node $G_{[r,s]}$ from the decomposition tree T. The main idea behind this algorithm is to leverage the sparsifier computed at the parent nodes as well as apply the efficient vertex sparsifiers from Theorem 6.7.2 Part 1. The procedure accomplishing this task for a single node of the

tree T is formally given in Algorithm 6.7. To compute the vertex sparsifier for every node, we simply apply it in a top-down fashion to the nodes of T.

Algorithm 6.7: VERTEXSPARSIFY($G_{[r,s]}$)

if $G_{[r,s]}$ *is the root node* **then**
  $G''_{[r,s]} = G'_{[r,s]} \leftarrow (V, \emptyset)$, i,e, the empty graph.
else
  Let $G_{[q,t]}$ be the parent of $G_{[r,s]}$ in T
  $G''_{[r,s]} \leftarrow \left(G'_{[q,t]} \cup H_{[r,s]}\right)$, where $G'_{[q,t]}$ is an efficient vertex sparsifier of $G_{[q,t]}$ with respect to $\partial_{[q,t]}$
  Let $G'_{[r,s]}$ be an α-efficient vertex sparsifier of $G''_{[r,s]}$ with respect to $\partial_{[r,s]}$ (Theorem 6.7.2 Part 1)
return $G'_{[r,s]}$

To argue about the usefulness of Algorithm 6.7, we need to bound the quality of sparsifiers produced at the nodes of T. The lemma below show that the quality grows multiplicatively with the number of levels in T.

Lemma 6.7.4. *Let $G_{[r,s]}$ be a node of T at level $i \geq 0$. Then $G' =$ VERTEXSPARSIFY($G_{[r,s]}$) outputs an α^i-efficient vertex sparsifier of $G_{[r,s]}$ with respect to $\partial_{[r,s]}$*

Proof. We proceed by induction on i. For the base case, i.e., $i = 0$, $G_{[1,m]}$ is the root node. Since $E^p_{[1,m]} = \emptyset$ by definition of permanent edges, we get that $G'_{[1,m]} = G_{[1,m]}$, i.e., $G_{[1,m]}$ is a sparsifier of itself.

Let $G_{[r,s]}$ be a node at level $i > 0$. Let $G_{[q,t]}$ be the parent of $G_{[r,s]}$ in T, and let $G'_{[q,t]}$ be its cut sparsifier at level $(i-1)$, as defined in Algorithm 6.7. By Property (4) of T note that $E^p_{[q,t]} \subseteq E^p_{[r,s]}$ since $[r,s] \subseteq [q,t]$. Also recall that $H_{[r,s]} = G_{[r,s]} \backslash G_{[q,t]}$. By induction hypothesis, we know that $G'_{[q,t]}$ is an α^{i-1}-efficient vertex sparsifier of $G_{[q,t]}$ with respect to $\partial_{[q,t]}$. This together with the decomposability property in Theorem 6.7.2 Part 2 imply that that $G''_{r,s} = G'_{[q,t]} \cup (G_{[r,s]} \setminus G_{[q,t]})$ is an α^{i-1}-efficient vertex sparsifier of $G_{[q,t]} \cup (G_{[r,s]} \backslash G_{[q,t]}) = G_{[r,s]}$ with respect to $\partial_{[q,t]}$. Now, by Theorem 6.7.2 Part 1 we get that $G'_{[r,s]}$ is an α-efficient vertex sparsifier of $G''_{[r,s]}$ with respect to $\partial_{[r,s]}$. Since $\partial_{[r,s]} \subseteq \partial_{[q,t]}$, and applying the transitivity property (Theorem 6.7.2 Part 2) on $G'_{[r,s]}$ and $G''_{[r,s]}$, we get that $G'_{[r,s]}$ is an $\alpha^{i-1+1} = \alpha^i$-efficient vertex sparsifier of $G_{[r,s]}$. □

We now state a crucial property of the nodes in the decomposition tree T, which allows us to get a reasonable bound on the running time for computing vertex sprasifiers for the nodes in T.

Lemma 6.7.5. *Let $G_{[r,s]}$ be a node in the decomposition tree T, and let $G_{[q,t]}$ be its parent. Then we have that the number of new permanent edges of $G_{[r,s]}$ is bounded by the number of non-permanent edges of its parent, i.e., $|E\left(H_{[r,s]}\right)| \leq |E^{np}_{[q,t]}|$.*

Proof. If an edges in in $H_{[r,s]}$, then it is not in $G^{p}_{[r,s]}$, thus it is a non-permanent edge in $G_{[q,t]}$. □

The lemma below gives a bound on the running time for computing vertex sparsifers in T.

Lemma 6.7.6. *The total running time for computing the vertex sparsifiers for each node in the decomposition tree T of height ℓ is bounded by*

$$\tilde{O}\left(\beta_0 \cdot \left(\sum_{j=1}^{\ell} \frac{\beta_{j-1}}{\beta_j}\right)\right), \quad \textit{where } \beta_0 = m.$$

Proof. For $i \geq 1$, let $Y(i)$ be the total time for computing the vertex sparsifiers for all the nodes in T up to (and including) level i. Furthermore, let $Z(i)$ be the total time for computing the vertex sparsifier of the nodes at level i in Y (and excluding other levels). We will show by induction on the number of levels i that $T(i) = O\left(\beta_0 \cdot \left(\sum_{j=1}^{i} \frac{\beta_{j-1}}{\beta j}\right) f(n)g(n)\right)$, which with $i = k$ implies the claim we want to prove.

For the base case, i.e., $i = 1$, consider any node $G_{[r,s]}$ at level 1 of T. By construction of T, $G_{[r,s]}$ contains at most $O(\beta_0)$ permanent edges. Furthermore, note that the parent of $G_{[r,s]}$ is the root node $G_{[1,m]}$, for which $G'_{[1,m]} = (V, \emptyset)$. Thus, by Theorem 6.7.2 Part 1 we get that the time to compute an efficient vertex sparsifier per node is $O(\beta_0 \cdot f(n))$. By Property (7) of T, the number of nodes at level 1 is $O(\beta_0/\beta_1)$, implying that the total running time is $Y(1) = Z(1) = O\left(\beta_0 \left(\frac{\beta_0}{\beta_1}\right) f(n)\right) = O\left(\beta_0 \left(\frac{\beta_0}{\beta_1}\right) f(n)g(n)\right)$, as desired.

We next show the inductive step. Let $G_{[r,s]}$ be a node at level $i > 1$, and let $G_{[q,t]}$ be its parent. We want to bound the size of the intermediate graph $G''_{[r,s]} = (G'_{[q,t]} \cup H_{[r,s]})$, as defined in Algorithm 6.7, which in turn determines the running time for computing an efficient vertex sparsifier of $G_{[r,s]}$. To this end, first observe that Theorem 6.7.2 Part 1 implies that the size of sparsifier $G'_{[q,t]}$ of $G_{[q,t]}$ is bounded by

$$O(|\partial_{[q,t]}| \cdot g(n)) \leq |V(E^{np}_{[r,s]}) \cup V(Q_{[r,s]})| \cdot g(n) \leq O(\beta_{i-1} \cdot g(n)),$$

since the number of non-permanent edges and queries is proportional to the length of the interval being considered. Second, by Lemma 6.7.5, we also have that $|E(H_{[r,s]})| \leq |E^{np}_{q,t}| \leq O(\beta_{i-1})$, thus giving that $|G''_{[r,s]}| \leq O(\beta_{i-1} \cdot g(n))$. As Algorithm 6.7 runs CutSparsify on the graph $G''_{[r,s]}$, Theorem 6.7.2 Part 1 gives that the running time to compute an efficient vertex sparsifier for the node $G_{[r,s]}$ is $O(\beta_{i-1} \cdot f(n)g(n))$, and that its size is $O(\beta_{i-1} \cdot g(n))$. Combining this together with the fact that the number of nodes at level i is at most $O(\beta_0/\beta_i)$ (Property (7) of T) imply that

$$Z(i) = O\left(\beta_0 \cdot \frac{\beta_{i-1}}{\beta_i} f(n)g(n)\right).$$

To complete the inductive step, note that by induction hypothesis,

$$Y(i-1) = O\left(\beta_0 \cdot \left(\sum_{j=1}^{i-1} \frac{\beta_j - 1}{\beta_j}\right) f(n)g(n)\right).$$

Summing over this and the bound on $Z(i)$ we get

$$\begin{aligned} Y(i) &= Y(i-1) + Z(i) \\ &= O\left(\beta_0 \cdot \left(\sum_{j=1}^{i-1} \frac{\beta_{j-1}}{\beta_j}\right) f(n)g(n)\right) + O\left(\beta_0 \cdot \left(\frac{\beta_{i-1}}{\beta_i}\right) f(n)g(n)\right) \\ &= O\left(\beta_0 \cdot \left(\sum_{j=1}^{i} \frac{\beta_{j-1}}{\beta j} f(n)g(n)\right)\right). \end{aligned}$$

□

Processing operations in the hierarchy. So far we have shown how to reduce the sequence of operations into smaller intervals in a hierarchical manner, while (approximately) preserving the properties of the edges and queries involved in the offline sequence. In what follows, we observe that for processing these events, it is sufficient to process the nodes (and their corresponding intervals) stored at the last level ℓ of the tree decomposition T (note that this is possible because intervals at level ℓ form a partitioning of the event sequence $[1, m]$, and all vertex pairs within intervals that will be involved in edge updates or queries are preserved using vertex sparsifiers).

The algorithm for processing the updates is quite simple: for every node $G_{[r,s]}$ at level ℓ of T, we process all operations in the interval consecutively: for each edge insertion or deletion we add or remove that suitable edges to $G'_{[r,s]}$, and for each query (x, y) we run on the vertex sparsifier $G'_{[r,s]}$ the static algorithm from Theorem 6.7.2 Part 3 to calculate the property $\mathcal{P}(x, y, G'_{[r,s]})$ between x and y in $G'_{[r,s]}$. (note that this is possible since $\partial_{[r,s]} \supseteq \{x, y\}$ by construction of T).

We next analyze the total time for processing the sequence of events in the last level of T.

Lemma 6.7.7. *The total time for processing the whole sequence of operations at level ℓ of the decomposition tree T is $\tilde{O}(\beta_0\beta_\ell \cdot g(n)h(n))$, where $\beta_0 = m$.*

Proof. As in the worst-case there can be at most $O(\beta_\ell)$ queries within the interval, and since the size of $G'_{[r,s]}$ is also bounded by $O(\beta_\ell g(n))$, by Theorem 6.7.2 Part 3 it follows that answering all the queries and processing the non-permanent edges within a single interval at level ℓ is bounded by $\tilde{O}(\beta_\ell^2 g(n)h(n))$. Combining this with the fact that the number of nodes at level ℓ is $O(\beta_0/\beta_\ell)$ (Property (7) of T), we get that the total cost for processing the queries is $\tilde{O}(\beta_0\beta_\ell \cdot g(n)h(n))$. □

Combining Lemma 6.7.6 and Lemma 6.7.7 leads to an overall performance of

$$\tilde{O}\left(\beta_0\left(\sum_{j=1}^{\ell}\left(\frac{\beta_{j-1}}{\beta_j}\right)f(n)+\beta_\ell h(n)\right)g(n)\right) \quad \text{where } \beta_0 = m,$$

which proves the claimed total update time in Theorem 6.7.2.

We finally prove the correctness of our algorithm. Concretely, we show that the estimate we return when processing any query (x, y) in the last level of the hierarchy approximates the property $\mathcal{P}$ of the graph G up to an α^ℓ factor, thus proving the claimed estimate in Theorem 6.7.2.

To this end, let q_i be a query in the sequence of operations $[1, m]$. Since the intervals at level ℓ of T form a partitioning of $[1, m]$, there must exist an interval $[r, s]$ that contains the query q_i. Let (x, y) be the queried vertex pair of q_i. By Lemma 6.7.4, we get that the graph $G'_{[r,s]}$ at level ℓ is an α^ℓ-vertex sparsifier of $G_{[r,s]}$ with respect to $\partial_{[r,s]}$. Since by construction $\partial_{[r,s]} \supseteq \{x, y\}$, we get that the $G'_{[r,s]}$ approximates the property $\mathcal{P}(x, y, G)$ of $G_{[r,s]}$ up to an α^ℓ factor. Finally, recall that we run the algorithm from Theorem 6.7.2 Part 3 on $G'_{[r,s]}$, thus worsening the approximation in the worst-case by at most a constant factor, which yields the claimed bound.

6.7.1 Applications to Offline Shortest Paths and Max Flow

In this section we show how to use our general Theorem 6.7.2 to design offline dynamic algorithms for the approximate All Pair Shortest Paths and All Pair Max Flow with reasonably small total update time.

We first consider shortest paths. Recall that our goal is to show that assumptions (1), (2) and (3) from Theorem 6.7.2 are satisfied with certain parameters for the shortest path measure. For (1) we make the following observation: given a graph G, a subset of terminals K, and a parameter $r \geq 1$, we can construct an *efficient* (vertex) distance sparsifier $(H, (2r-1), \tilde{O}(n^{1/r}), \tilde{O}(n^{1/r}))$ by simply constructing an efficient local sparsifier for G using Lemma 6.4.4 and querying it with respect to K. Also note that assumption (2) is satisfied by the transitivity and decomposability of H, and finally recall that (3) follows by any $\tilde{O}(m)$ time single pair shortest path algorithm. These together imply the following result.

Theorem 6.7.8. *Let $G = (V, E)$ be an undirected, weighted graph. For every $r, \ell \geq 1$, there is an offline fully dynamic approximate All Pair Shortest Path algorithm that maintains for every pair of nodes u and v, a distance estimate $\delta(u, v)$ such that*

$$\mathit{dist}_G(u, v) \leq \delta(u, v) \leq (2r-1)^\ell \mathit{dist}_G(u, v).$$

The total time for processing a sequence of m operations is

$$\tilde{O}(m \cdot m^{1/(\ell+1)} n^{2/r}).$$

We now proceed with max flow. Following essentially the same idea as with shortest paths, we need to show that assumptions (1), (2) and (3) from Theorem 6.7.2 are satisfied with certain parameters for the max flow measure. For (1) we have the following: given a graph G, a subset of terminals K, we can construct an *efficient* (vertex) flow sparsifier $(H, O(\log^4 n), \tilde{O}(1), \tilde{O}(1))$ by simply constructing an efficient flow local sparsifier for G using Lemma 6.5.4 and querying it with respect to K. Also note that assumption (2) is satisfied by the transitivity and decomposability of H, and finally recall that (3) follows by employing the $\tilde{O}(m)$ time (approximate) (s,t)-maximum flow algorithm due to Peng [210]. These together imply the following theorem.

Theorem 6.7.9. *Let $G = (V, E)$ be an undirected, weighted graph. For every $\ell \geq 1$, there is an offline fully dynamic approximate All Pairs Max Flow algorithm that maintains for every pair of nodes u and v, a flow estimate $\delta(u, v)$ such that*

$$\frac{1}{\tilde{O}(\log^{4\ell} n)} \text{max-flow}_G(u, v) \leq \delta(u, v) \leq \text{max-flow}_G(u, v).$$

The total time for processing a sequence of m operations is

$$\tilde{O}(m \cdot m^{1/(\ell+1)}).$$

6.8 Implications on Hardness of Approximate Dynamic Problems

6.8.1 Approximate max flow and cut sparsifiers

Assuming the OMv conjecture, Dahlgaard [78] show that any incremental exact max flow algorithm on undirected graphs must have amortized update time at least $\Omega(n^{1-o(1)})$. However, the hardness of approximation is not known[7]:

Proposition 6.8.1. *There is no polynomial lower bound for dynamic $\omega(\text{polylog}(n))$-approximate max flow in the offline setting (and also in the online incremental setting).*

This follows directly from Theorems 6.5.1 and 6.7.9. Thus the important open problem is whether we can prove a hardness for dynamic $(1+\epsilon)$-approximate max flow algorithms on undirected graphs for a constant $\epsilon > 0$.

On the other hand, it is not known whether, given a set of k terminals, there is a $(1+\epsilon)$-approximate cut (vertex) sparsifier of size $\text{poly}(k, 1/\epsilon)$ or even $\text{poly}(k, 1/\epsilon, \log n)$. If a cut sparsifier can only contain terminals as nodes, then the approximation ratio must be at least $\Omega(\sqrt{\log k}/\log\log k)$ [191]. If we need an exact cut sparsifier, then the size must be at least $2^{\Omega(k)}$ [172].

[7] However, on directed graphs, the hardness of approximation is known. This is because even dynamic reachability is hard under several conjectures [6, 135].

In what follows we draw a connection between these two open problems; if there is a very efficient algorithm for the above cut sparsifier, then there cannot be a $\Omega(n^{1-o(1)})$ lower bound in the offline setting for the dynamic approximate max flow. Moreover, if the cut-sparsifier has size almost best possible, then there cannot be even a super-polylogarithmic lower bound. Concretely, we show the following.

Theorem 6.8.2. *If there is an algorithm that, given a undirected graph $G = (V, E)$ with m edges and a set $T \subset V$ of k terminals, constructs an $(1+\epsilon)$-approximate cut vertex sparsifier of size $s = \text{poly}(k, 1/\epsilon, \log n)$ in time $O(m \,\text{poly}(\log n, 1/\epsilon))$, there is an offline dynamic algorithm for maintaining $(1+\epsilon')$-approximate value of max flow with update time $u = O(n^{1-\gamma}\,\text{poly}(1/\epsilon'))$ for some constant $\gamma > 0$. Moreover, if the size of the sparsifier $s = k \cdot \text{poly}(1/\epsilon, \log n)$, then we obtain the update time of $u = O(\text{poly}(\log n, 1/\epsilon'))$. The dynamic algorithm is Monte Carlo randomized and it is correct with high probability.*

Proof. Let us assume ϵ' is a constant for simplicity. The proof generalizes easily when ϵ' is not a constant.

First, we only need to consider offline dynamic algorithms where the underling graph has $m = \tilde{O}(n)$ edges at every time step and the length of the update sequences is n. This is because there is a dynamic algorithm by [12] that can maintain a cut sparsifier $H = (V, E')$ of a graph $G = (V, E)$ when the terminal set is V with $\tilde{O}(1)$ worst-case update. So we can work on H instead, and divide the update sequences into segements of length n. If we have an offline dynamic algorithm with update time u on average on each period, then the average update time is $\tilde{O}(u)$ over the whole sequence.

We set $\epsilon = \epsilon'/10 \log n$. Suppose that the sparsifier from the assumption has size only $s = k \cdot \text{poly}(1/\epsilon, \log n) = \tilde{O}(k)$. Then, we apply the same proof as in Theorem 6.7.9, except that the number of levels of the decomposition tree will be $\log n$ instead of $O(\sqrt{\log n})$. The quality of the cut-sparsifier at any level is at most $(1+\epsilon)^{\log n} = (1 + \epsilon'/10 \log n)^{\log n} \leq (1+\epsilon')$. The total running time will be $\tilde{O}(m^{1+\frac{1}{\log n+1}}) = \tilde{O}(n)$. The latter implies that update time on average is $O(\text{polylog}(n))$.

Assume that $s = k^c \cdot \text{poly}(1/\epsilon, \log n) = \tilde{O}(k^c)$ for some constant $c > 1$. Then, we can apply again the same proof from Theorem 6.7.9. By using only two levels of the decomposition tree, we can obtain an update time of $\tilde{O}(n^{1-\frac{1}{c+1}})$. Concretely, if we set $\beta_0 = m$ and $\beta_1 = m^{1/(c+1)}$ then the time for computing the decomposition tree is $\frac{\beta_0}{\beta_1} \cdot \tilde{O}(\beta_0) = \tilde{O}(n^{2-\frac{1}{c+1}})$. The total time for running approximate max flow on the cut-sparsifier in the second level at each step is $\beta_0 \cdot \tilde{O}(\beta_1^c) = \tilde{O}(n^{2-\frac{1}{c+1}})$. Thus it follows that the update time is $\tilde{O}(n^{1-\frac{1}{c+1}})$ on average. □

6.8.2 Approximate distance oracles on general graphs

There are previous hardness results for approximation algorithms for dynamic shortest path problems (including single-pair, single-source and all-pairs problems) [135].

All such results show a very high lower bound, e.g. $\Omega(n^{1-\epsilon})$ or $\Omega(n^{1/2-\epsilon})$ time on an n-node graph. However, they hold only when the approximation factor is a small constant. It is open whether one can obtain weaker polynomial lower bounds for larger approximation factors. We show that it is impossible to show super-constant factor lower-bounds in several settings.

Proposition 6.8.3. *There is no polynomial lower bound for dynamic $\omega(1)$-approximate distance oracles in the offline setting (and also in the online incremental setting).*

More formally, for any lower bound stating that $\omega(1)$-approximate offline dynamic distance oracle algorithm on n-node graphs requires at least $u(n)$ update time or $q(n)$ query time, then we have $u(n) = n^{o(1)}$ and $q(n) = n^{o(1)}$. The same holds for online incremental algorithm with worst-case update time.

This follows directly from Theorems 6.4.1 and 6.7.8.

6.8.3 Approximate distance oracles on planar graphs

Similar to the situations above, assuming the APSP conjecture, Abboud and Dahlgaard [4] show that any offline fully dynamic algorithm for exact distance oracles on planar graph requires either update time or query time of $\Omega(n^{1/2-o(1)})$. We can still hope for a hardness result for $(1+\epsilon)$-approximate distance oracles, but this remains an important open problem in the field of dynamic algorithms.

Recall the definition of *distance approximating minors* from Chapter 7, which are vertex distance sparsifiers that are required to be minors of the input graph. In the exact setting, Krauthgamer et al. [170] showed that any distance preserving minor with respect to k terminals, even when restricted to planar graphs, must have size $\Omega(k^2)$ size. Cheung et al. [69] showed that for planar graphs there is a $(1+\epsilon)$-distance approximating minor of size $\tilde{O}(k^2\epsilon^{-2})$. The natural question is whether there is a $(1+\epsilon)$-approximate *minor* distance sparsifier for k terminals that has size $k^{1.99} \cdot \text{poly}(1/\epsilon, \log n)$.

We again draw a connection between dynamic graph algorithms and vertex sparsifiers; if there is a very efficient algorithm for such distance sparsifiers, then we cannot extend the $\Omega(n^{1/2-o(1)})$ lower bound to the approximate setting. Moreover, if the sparsifier has the (almost) best possible size, then there cannot be even a super-polylogarithmic lower bound. More precisely, we show the following.

Theorem 6.8.4. *Let G be an undirected graph $G = (V, E)$ with m edges and a set $K \subset V$ of k terminals. If there is an algorithm that constructs a $(1+\epsilon)$-distance approximating minor of size $s = k^{2/(1+3\gamma)} \cdot \text{poly}(1/\epsilon, \log n)$, for some constant $0 < \gamma \leq 1/3$, in time $O(m\,\text{poly}(\log n, 1/\epsilon))$, then there is an offline dynamic $(1+\epsilon')$-approximate distance oracle algorithm for with update and query time $u = O(n^{1/2-\gamma/2})$. In fact, if the size of the sparsifier is $s = k \cdot \text{poly}(1/\epsilon', \log n)$, then we obtain an update and query time of $u = O(\text{poly}(\log n))$.*

The proof will be very similar to the one in Theorem 6.8.2 except that we need to be more careful about planarity. Thus we first proving the following useful lemma.

Lemma 6.8.5. *Each vertex sparsifier $G'_{[r_p,s_p]}$ corresponding to a node in our decomposition tree is planar.*

Proof. First, consider a sequence of $H_{[r_1,s_1]}, H_{[r_2,s_2]}, \ldots, H_{[r_p,s_p]}$ corresponding to a path in the decomposition tree, where $H_{[r_1,s_1]}$ is a child of the root[8], and $H_{[r_i,s_i]}$ is a parent of $H_{[r_{i+1},s_{i+1}]}$. Observe that $\cup_{1\leq i\leq p} H_{[r_i,s_i]} = G_{[r_p,s_p]}$ which is planar.

From Algorithm 6.7, we unfold the recursion and obtain that

$$G'_{[r_p,s_p]} = \textsc{VertexSparsify}(\textsc{VertexSparsify}(\ldots) \cup H_{[r_{p-1},s_{p-1}]}) \cup H_{[r_p,s_p]}).$$

Note that we omit the second parameter of VertexSparsify only for readability. We assume by induction $G'_{[r_{p-1},s_{p-1}]} = \textsc{VertexSparsify}(\textsc{VertexSparsify}(\ldots) \cup H_{[r_{p-1},s_{p-1}]})$ is planar. We will prove that $G'_{[r_p,s_p]}$ planar. To this end, observe that $G'_{[r_{p-1},s_{p-1}]}$ is a minor of $\cup_{1\leq i\leq p-1} H_{[r_i,s_i]}$. Next, we need the following observation.

Claim 6.8.6. *Let G_1 be a minor of G_2. Let (u,v) be an edge such that $u,v \in V(G_1) \cap V(G_2)$, i.e., the endpoints are nodes of both G_1 and G_2. Then, $G_1 \cup \{(u,v)\}$ is a minor of $G_2 \cup \{(u,v)\}$. In particular, if $G_2 \cup \{(u,v)\}(u,v)$ is planar, then so is $G_1 \cup \{(u,v)\}$.*

We apply Claim 6.8.6 where $G_2 = \cup_{1\leq i\leq p-1} H_{[r_i,s_i]}$ and $G_1 = G'_{[r_{p-1},s_{p-1}]}$. As the endpoints of $H_{[r_i,s_i]}$ are in both G_1 and G_2 by construction and $G_2 \cup H_{[r_p,s_p]} = \cup_{1\leq i\leq p} H_{[r_i,s_i]}$ is planar, then $G_1 \cup H_{[r_p,s_p]}$ is planar. Finally, $G'_{[r_p,s_p]} = \textsc{VertexSparsify}(G_1 \cup H_{[r_p,s_p]})$ is a minor of $G_1 \cup H_{[r_p,s_p]}$, so $G'_{[r_p,s_p]}$ is planar. □

Now, we prove Theorem 6.8.4.

Proof of Theorem 6.8.4. We first prove the case when $s = k \cdot \text{poly}(1/\epsilon, \log n)$. We again prove the theorem when ϵ' is a constant for simplicity. Set $\epsilon = \epsilon'/10\log n$. We build the corresponding decomposition tree with $\log n$ levels. The quality of the sparsifier at any level is at most $(1+\epsilon)^{\log n} = (1+\epsilon'/10\log n)^{\log n} \leq (1+\epsilon')$. The total running time will be $\tilde{O}(m^{1+\frac{1}{\log n+1}}) = \tilde{O}(n)$ using the same argument as in Lemma 6.3.4. That is the update time on average is $O(\text{poly}(\log n))$.

For the case when $s = k^{2/(1+3\gamma)} \cdot \text{poly}(1/\epsilon, \log n)$, the proof is the same except the parameters need to be carefully chosen. We set $\epsilon = \epsilon'\gamma/2$. We choose $\beta_0 = m = O(n)$, $\beta_1 = n^{(1+\gamma)/2}$, and $\beta_{i+1} = n^{(1+\gamma-2\gamma i)/2}$ for $i \geq 0$. We get that there will be at most $1/\gamma$ levels in the decomposition tree and thus the quality at each level is at most

$$(1+\epsilon)^{1/\gamma} \leq e^{\epsilon/\gamma} = e^{\epsilon'/2} \leq (1+\epsilon')$$

because $(1+x) \leq e^x$ for any x and $e^{x/2} \leq (1+x)$ for $0 \leq x \leq 1$.

[8]Note that the graph $H_{[r,s]}$ is not defined at the root.

For each i, the total time to build the sparsifiers in level $i+1$ by running the algorithm sparsifier at level i is $n/\beta_{i+1} \cdot \tilde{O}(\beta_i^{2/(1+3\gamma)})$. This is because there are n/β_{i+1} many sparsifiers, and the algorithm is applied on a graph of size $\tilde{O}(\beta_i^{2/(1+3\gamma)})$. By direct calculation we have that

$$n/\beta_{i+1} \cdot \beta_i^{2/(1+3\gamma)} = n^{1-(1+\gamma-2i\gamma)/2+\frac{(1+\gamma-2(i-1)\gamma)}{(1+3\gamma)}} \leq n^{1.5-\gamma/2}.$$

To see this, note that $2/(1+3\gamma) \geq 1$ and consider the following chain of inequalities:

$$\begin{aligned}
&\frac{(1+\gamma-2(i-1)\gamma)}{(1+3\gamma)} - (1+\gamma-2i\gamma)/2 \\
&\qquad \leq \frac{1+\gamma}{1+3\gamma} - (i-1)\gamma - \frac{1+\gamma}{2} + i\gamma \\
&\qquad \leq (1-\gamma) + \gamma - 1/2 - \gamma/2 \\
&\qquad = 1/2 - \gamma/2.
\end{aligned}$$

It follows that the total time over all levels is $\frac{1}{\gamma} \cdot O(n^{1.5-\gamma/2})$, which is turn implies an average update time of $O(n^{0.5-\gamma/2})$. This completes the proof. □

6.9 Conclusion

In this chapter, we showed a fast incremental algorithm for approximating all-pairs shortest paths, all-pairs max flow, multi-commodity concurrent flow and uniform sparsest cut. Our algorithmic constructions require poly-logarithmic approximation while achieving sub-linear time for all these problems, except shortest path, for which our approximate ratio improves to constant. The key building block behind our meta algorithm is a new sparsification notion, referred to as a local sparsifier, that generalizes the well-known notion of vertex sparsification. We also systemically study the power of (classic) vertex sparsification in the design of efficient offline dynamic algorithms, where the sequence of updates and queries is given beforehand.

Our work motivates the study of several important research directions. First, an important open problem is whether one can construct efficient local sparsifiers for cuts with constant quality, even when restricted to planar graphs. Recall that our construction uses trees and that there is a lower bound of $\Omega(\log n)$ on the quality when approximating the cut structure of a graph by a tree [215].

Second, an interesting problem is to construct efficient local sparsifiers for effective resistances. At first, this problem seems promising as there are already near-linear time construction of vertex resistance sparsifiers [88], known as approximate Schur complements. However, this construction employs approximate Gaussian elimination and thus it is highly sequential. It is worth investigating whether there are other ways of constructing such sparsifiers that would extend to the local setting.

CHAPTER 7

Graph Minors for Preserving Terminal Distances Approximately – Lower and Upper Bounds

Given a graph where vertices are partitioned into k terminals and non-terminals, the goal is to compress the graph (i.e., reduce the number of non-terminals) using minor operations while preserving terminal distances approximately. The distortion of a compressed graph is the maximum multiplicative blow-up of distances between all pairs of terminals. We study the trade-off between the number of non-terminals and the distortion. This problem generalizes the Steiner Point Removal (SPR) problem, in which all non-terminals must be removed.

We introduce a novel black-box reduction to convert any lower bound on distortion for the SPR problem into a super-linear lower bound on the number of non-terminals, with the same distortion, for our problem. This allows us to show that there exist graphs such that every minor with distortion less than 2, $5/2$ and 3 must have $\Omega(k^2)$, $\Omega(k^{5/4})$, and $\Omega(k^{6/5})$ non-terminals, respectively, plus more trade-offs in between. The black-box reduction has an interesting consequence: if the tight lower bound on distortion for the SPR problem is super-constant, then allowing any $O(k)$ non-terminals will *not* help improving the lower bound to a constant.

We also build on the existing results on spanners, distance oracles and connected 0-extensions to show a number of upper bounds for general graphs, planar graphs, graphs that exclude a fixed minor and bounded treewidth graphs. Among others, we show that any graph admits a minor with $O(\log k)$ distortion and $O(k^2)$ non-terminals, and any planar graph admits a minor with $1 + \epsilon$ distortion and $\tilde{O}(k^2\epsilon^{-2}\log^2 k)$ non-terminals.

7.1 Introduction

Graph compression generally describes a transformation of a *large* graph G into a *smaller* graph H that preserves, either exactly or approximately, certain features (e.g., distance, cut, flow) of G. Its algorithmic value is apparent, since the compressed graph can be computed in a preprocessing step of an algorithm, so as to reduce subsequent running time and memory. Some notable examples are graph spanners, distance oracles and cut/flow sparsifiers.

In this chapter, we study compression using minor operations, which has attracted increasing attention in recent years. Minor operations include vertex/edge deletions and edge contractions. It is naturally motivated since it preserves certain structural properties of the original graph, e.g., any minor of a planar graph remains planar, while reducing the size of the graph. We are interested in *vertex sparsification*, where G has a designated subset K of k vertices called the *terminals*, and the goal is to reduce the number of non-terminals in H while preserving some feature among the terminals. Recent work in this field studied preserving cuts and flows. Our focus here is on preserving terminal distances approximately in a multiplicative sense, i.e., we want that for any pairs of terminals $u, v \in K$, $\text{dist}_G(u,v) \leq \text{dist}_H(u,v) \leq \alpha \cdot \text{dist}_G(u,v)$, for a small *distortion* α. This problem, called *Approximate Terminal Distance Preservation (ATDP) problem*, has natural applications in multicast routing [71] and network traffic optimization [226]. It was also suggested in [170] that to solve the *subset travelling salesman problem*, one can compute a compressed minor with a small distortion as a preprocessing step for algorithms that solve the travelling salesman problem for planar graphs.

ATDP was initiated by Gupta [119], who introduced the related *Steiner Point Removal (SPR) problem*: Given a tree G with both terminals and non-terminals, output a weighted tree G' *with terminals only* which minimizes the distortion. Gupta gave an algorithm that achieves a distortion of 8. Chan et al. [58] observed that Gupta's algorithm returned always a minor of G. For general graphs, Kamma et al. [146] gave an algorithm to construct a minor with distortion $O(\log^5 k)$. This bound has been recently improved to $O(\log^2 k)$ by Cheung [68] and finally to $O(\log k)$ by Filtser [102]. Krauthgamer et al. [170] studied ATDP and showed that every graph has a minor with $O(k^4)$ non-terminals and distortion 1. It is then natural to ask, for different classes of graphs, what is the trade-off between the distortion and the number of non-terminals. In this chapter, for different classes of graphs, and with respect to different allowed distortions, we provide lower and upper bounds on the number of non-terminals needed.

Further Related Work. Basu and Gupta [30] showed that for outer-planar graphs, SPR can be solved with distortion $O(1)$. When randomization is allowed, Englert et al. [94] showed that for graphs that exclude a fixed minor, one can construct a randomized minor for SPR with $O(1)$ expected distortion. It remains open whether similar guarantees can be obtained in the deterministic setting.

Krauthgamer et al. [170] showed that solving ATDP with distortion 1 for planar graphs needs $\Omega(k^2)$ non-terminals.

In the past few years, there has been a considerable amount of work on vertex sparsifiers that preserve cuts [24, 60, 72, 94, 180, 191, 197, 216]. In this setting, the goal is to compress the graph only on the termials while approximately preserving all possible terminal minimum cuts. This problem is closely connected to distance sparsification, and there exist techniques to construct vertex sparsifiers for cuts using distance sparsifiers, and vice versa [94, 214].

A related graph compression is spanners, where the objective is to reduce the number of edges by edge deletions only. We will use a spanner algorithm (e.g., [20]) to derive our upper bound results for general graphs. Although spanner operation enjoys much less freedom than minor operation, proving a lower bound result for it is notably difficult. Assuming the Erdös girth conjecture [98], there are lower bounds that match the best known upper bounds, but the conjecture seems far from being settled [256]. Woodruff [259] showed a lower bound result bypassing the conjecture, but only for *additive* spanners.

Our Results. For various classes of graphs, we show lower and upper bounds on the number of non-terminals needed in the minor for low distortion. The table below summarizes our results. For our lower bound results, we use a novel black-box reduction to convert any lower bound on distortion for the SPR problem into a super-linear lower bound on the number of non-terminals for ATDP with the same distortion. Precisely, we show that given any graph G^* such that solving its SPR problem leads to a minimum distortion of α, we use G^* to construct a new graph G such that every minor of G with distortion less than α must have at least $\Omega(k^{1+\delta(G^*)})$ non-terminals, for some constant $\delta(G^*) > 0$. The lower bound results in the above table are obtained by using for G^* a complete ternary tree of height 2, which was shown that solving its SPR problem leads to minimum distortion 3 [119]. More trade-offs are shown by using for G^* a complete ternary tree of larger heights.

The black-box reduction has an interesting consequence. For the SPR problem on general graphs, there is a huge gap between the best known lower and upper bounds, which are 8 [58] and $O(\log k)$ [102]; it is unclear what the asymptotically tight bound would be. Our black-box reduction allows us to prove the following result concerning the tight bound: for general graphs, if the tight bound on distortion for the SPR problem is super-constant, then for any constant $c > 0$, even if ck non-terminals are allowed in the minor, the lower bound will remain super-constant. See Theorem 7.3.13 for a formal statement of this result.

We also build on the existing results on spanners, distance oracles and connected 0-extensions to show a number of upper bound results for general graphs, planar graphs and graphs that exclude a fixed minor. Our techniques, combined with an algorithm in Krauthgamer et al. [170], yield an upper bound result for graphs with bounded treewidth. In particular, our upper bound on planar graphs implies that allowing quadratic number of non-terminals, we can construct a deterministic minor

Graph	Upper Bound	Lower Bound
	(distortion, size)	(distortion, size)
General	$(2q-1, O(k^{2+2/q}))$	$(2-\varepsilon, \Omega(k^2))$
General	–	$(2.5-\varepsilon, \Omega(k^{5/4}))$
		$(3-\varepsilon, \Omega(k^{6/5}))$
		(see Theorem 7.3.6 for more guarantees)
B.-Treewidth p	$(2q-1, O(p^{1+2/q}k))$	$(1, \Omega(pk))$ [170]
Exc.-Fix.-Minor	$(O(1), \tilde{O}(k^2)$	–
Planar	$(1+\varepsilon, \tilde{O}((k/\varepsilon)^2)$	$(1+o(1), \Omega(k^2))$ [170]
General	$(O(\log k), 0)$ [102]	–
Outerplanar	$(O(1), 0)$ [30]	–
Trees	$(8, 0)$ [119]	$(8-o(1), 0)$ [58]
General	$(O(\log k), 0)$-rand [94]	–
Exc.-Fix.-Minor	$(O(1), 0)$-rand [94]	$(2-o(1), 0)$-rand

Table 7.1: The results which are *not* followed by a reference are shown in this chapter. The guarantees with the extension "-rand" refer to *randomized* distance approximating minors; "size" refers to the number of non-terminals in the minor.

with arbitrarily small distortion.

7.2 Preliminaries

Let $G = (V, E, \mathbf{w})$ denote an undirected graph with terminal set $K \subset V$ of cardinality k, where $\mathbf{w} : E \to \mathbb{R}^+$ is the weight (length) function over edges E. A graph H is a *minor* of G if H can be obtained from G by performing a sequence of vertex/edge deletions and edge contractions, but no terminal can be deleted, and no two terminals can be contracted together. In other words, all terminals in G must be *preserved* in H.

Besides the above standard description of minor operations, there is another equivalent way to construct a minor H from G [146], which will be more convenient for presenting some of our results. A partial partition of $V(G)$ is a collection of pairwise disjoint subsets of $V(G)$ (but their union can be a proper subset of $V(G)$). Let $S_1, \cdots, S_m$ be a partial partition of $V(G)$ such that (1) each induced graph $G[S_i]$ is connected, (2) each terminal belongs to exactly one of these partial partitions, and (3) no two terminals belong to the same partial partition. Contract the vertices in each S_i into one single "super-node" in H. For any vertex $u \in V(G)$, let $S(u)$ denote the partial partition that contains u; for any super-node $u \in V(H)$, let $S(u)$

denote the partial partition that is contracted into u. In H, super-nodes u_1, u_2 are adjacent *only if* there exists an edge in G with one of its endpoints in $S(u_1)$ and the other in $S(u_2)$. We denote the super-node that contains terminal u by u as well.

Definition 7.2.1. *The graph $H = (V', E', \mathbf{w}')$ is an α-distance approximating minor (abbr. α-DAM) of $G = (V, E, \mathbf{w})$ if H is a minor of G and for any $u, v \in K$, $dist_G(u, v) \leq dist_H(u, v) \leq \alpha \cdot dist_G(u, v)$. H is an (α, y)-DAM of G if H is an α-DAM of G with at most y non-terminals.*

We note that the SPR problem is equivalent to finding an $(\alpha, 0)$-DAM. One can also define a randomized version of distance approximating minor:

Definition 7.2.2. *Let η be a probability distribution over minors of $G = (V, E, \mathbf{w})$. We call η an α-randomized distance approximating minor (abbr. α-rDAM) of G if for any $u, v \in K$,*

$$\mathbb{E}_{H \sim \eta} [dist_H(u, v)] \leq \alpha \cdot dist_G(u, v),$$

and for every minor H in the support of η, $dist_H(u, v) \geq dist_G(u, v)$. Furthermore, we call η an (α, y)-rDAM if η is an α-rDAM of G, and every minor in the support of η has at most y non-terminals.

7.3 Deterministic Lower Bounds

For all the lower bound results, we use a tool in combinatorial design called *Steiner system* (or alternatively, *balanced incomplete block design*). Let $[k]$ denote the set $\{1, 2, \cdots, k\}$.

Definition 7.3.1. *Given a ground set $K = [k]$, an $(s, 2)$-Steiner system (abbr. $(s, 2)$-SS) of K is a collection of s-subsets of K, denoted by $\mathcal{K} = \{K_1, \cdots, K_r\}$, where $r = \binom{k}{2} / \binom{s}{2}$, such that every 2-subset of K is contained in exactly one of the s-subsets.*

Lemma 7.3.2 ([258])**.** *For any integer $s \geq 2$, there exists an integer M_s such that for every $q \in \mathbb{N}$, the set $[M_s + qs(s-1)]$ admits an $(s, 2)$-SS.*

Our general strategy is to use the following black-box reduction, which proceeds by taking a *small* connected graph G^* as input, and it outputs a *large* graph G which contains many disjoint embeddings of G^*. Here is how it exactly proceeds:

- Let G^* be a graph with $s \geq 2$ terminals and $q \geq 1$ non-terminals. Let k be an integer, as given in Lemma 7.3.2, such that the terminal set $K = [k]$ admits an $(s, 2)$-SS $\mathcal{K}$.
- We construct $\mathcal{K}' \subseteq \mathcal{K}$ that satisfies *certain* property depending on the specific problem. For each s-set in $\mathcal{K}'$, we add q non-terminals to the s-set, which altogether form a *group*. The union of vertices in all groups is the vertex set of our graph G. We note that each terminal may appear in many groups, but each non-terminal appears in one group only.

- *Within* each of the groups, we embed G^* in the natural way.

The following two lemmas describe some basic properties of all minors of G output by the black-box above. Before presenting their proofs, we need to introduce some helpful notation. Let G be an output graph from the black-box. In any minor H of G, we say a super-node is of Type-A if $S(u)$ contains only non-terminals in G; any other super-node u, for which $S(u)$ contains *exactly* one terminal, is of Type-B. Here are two simple facts:

(a) If u is of Type-A, since $G[S(u)]$ is connected, the non-terminals in $S(u)$ must belong to the same group.

(b) If u is of Type-B, let t be the terminal in $S(u)$. If $S(u)$ contains a vertex from some group R, then $t \in R$.

Lemma 7.3.3. *Let H be a minor of G. Then for each edge (u_1, u_2) in H, there exists exactly one group R in G such that $S(u_1) \cap R$ and $S(u_2) \cap R$ are both non-empty.*

Proof. Existence of R is easy to prove by a simple induction on the minor operation sequence that generates H from G. To show the uniqueness, we proceed to a case analysis. In the first case, either u_1 or u_2 is of Type-A. Then the uniqueness is trivial by fact (a).

In the second case, both u_1, u_2 are of Type-B. For $i = 1, 2$, let v_i be the terminal in $S(u_i)$. Suppose there are two groups R_a, R_b that intersect both $S(u_1)$ and $S(u_2)$. Then by fact (b), v_1, v_2 are in both R_a and R_b, a contradiction. □

The above lemma permits us to legitimately define the notion R-edge: an edge (u_1, u_2) in H is an R-edge if R is the unique group that intersects both $S(u_1)$ and $S(u_2)$.

Lemma 7.3.4. *Suppose that in a minor H of G, (u_1, u_2) is a R_1-edge and (u_2, u_3) is R_2-edge, where $R_1 \neq R_2$. Then R_1 and R_2 intersect, and $S(u_2)$ contains the terminal in $R_1 \cap R_2$.*

Proof. Since $S(u_2)$ contains vertices from both R_1 and R_2, u_2 must be of Type-B, i.e., $S(u_2)$ contains exactly one terminal v. By fact (b), v is in both R_1 and R_2. □

We will show that for any minor H with low distortion, at least one of the non-terminals in each group must be retained, and thus H must have at least $|\mathcal{K}'|$ non-terminals. We now present our main theorems on lower bounds and then prove them.

Theorem 7.3.5. *For infinitely many $k \in \mathbb{N}$, there exists a bipartite graph with k terminals which does not have a $(2 - \epsilon, k^2/7)$-DAM, for all $\epsilon > 0$.*

Theorem 7.3.6. *There exists a constant $c_1 > 0$, such that for infinitely many $k \in \mathbb{N}$, there exists a quasi-bipartite graph with k terminals which does not have an $(\alpha - \epsilon, c_1 k^\gamma)$-DAM, for all $\epsilon > 0$, where α, γ are given in the table below.*

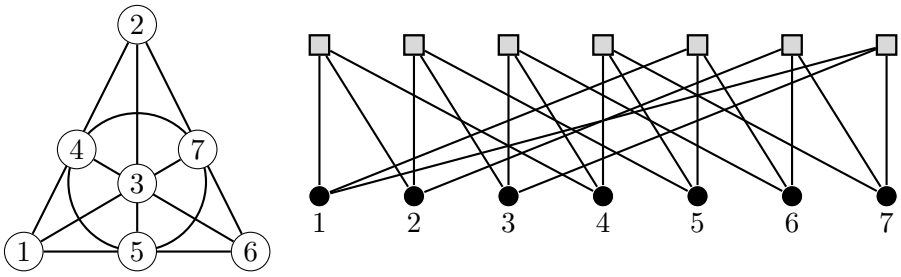

Figure 7.1: On the left side: a Fano plane corresponding to a $(3, 2)$-SS with $k = 7$. On the right side: the bipartite graph of the Fano plane constructed using our black-box reduction. Numbered vertices are *terminals* while square-shaped vertices are *non-terminals*.

α	2.5	3	10/3	11/3	4	4.2	4.4
γ	5/4	6/5	10/9	11/10	12/11	21/20	22/21

7.3.1 Distortion 2 Lower Bound

We next prove Theorem 7.3.5. Let us start by reviewing the lower bound for SPR problem on stars due to Gupta [119].

Lemma 7.3.7. *Let $G^* = (K \cup \{v\}, E)$ be an unweighted star with $k \geq 3$ terminals, in which v is the center of the star. Then, every edge-weighted graph only on the terminals K with fewer than $\binom{k}{2}$ edges has distortion at least 2.*

We construct G using the black-box reduction above. Let $k \in \mathbb{N}$ be such that the terminals $K = [k]$ admits a $(3, 2)$-SS, denoted by $\mathcal{K}$ (see the figure above). Here, we set $\mathcal{K}' = \mathcal{K}$ and G^* to be the star with 3 terminals, as described in Lemma 7.3.7.

By the definition of Steiner system, the shortest path between every pair of terminal u, v in G is unique, which is the 2-hop path within the group that contains both terminals, i.e., $\text{dist}_G(u, v) = 2$ for all $u, v \in K$. Every other simple path between u, v must pass through an extra terminal, so the length of such simple path is at least 4.

Let H be a minor of G. Suppose that the number of non-terminals in H is less than r, then there exists a group R in which its non-terminal is not retained (which means that it is either deleted, or contracted into a terminal in that group). By Lemma 7.3.7, there exists a pair of terminals in that group such that every simple path within R (which means a path comprising of R-edges only) between the two terminals has length at least 4. And every other simple path must pass through an extra terminal (just as in G), so again it has length at least 4. Thus, the distortion of the two terminals is at least 2.

Therefore, every $(2 - \epsilon)$-DAM of G must have $r > k^2/7$ non-terminals.

7.3.2 Higher Distortion Lower Bounds

We now prove Theorem 7.3.6. Concretely, we will give the proof for the case $\alpha = 2.5$ here, and discuss how to generalize this proof for other distortions. We will first define the notions of *detouring graph* and *detouring cycle*, and then use them to construct the graph G that allows us to show the lower bound.

Detouring Graph and Detouring Cycle. For any $s \geq 3$, let $k \in \mathbb{N}$ be such that the terminal set $K = [k]$ admits an $(s, 2)$-SS. Let $\mathcal{K} = \{K_1, \cdots, K_r\}$ be such an $(s, 2)$-SS. A *detouring graph* has the vertex set $\mathcal{K}$. By the definition of Steiner system, $|K_i \cap K_j|$ is either zero or one. In the detouring graph, K_i is adjacent to K_j if and only if $|K_i \cap K_j| = 1$. Thus, in the detouring graph, it is legitimate to give each edge (K_i, K_j) a *terminal label*, which is the terminal in $K_i \cap K_j$. A *detouring cycle* is a cycle in the detouring graph such that no two neighboring edges of the cycle have the same terminal label.

Fact 7.3.8. *Suppose that two edges in the detouring graph have a common vertex, and their terminal labels are different, denoted by u, v. Then the common vertex must be an s-set in $\mathcal{K}$ containing both u, v. By the definition of Steiner system, the s-set is uniquely determined.*

Claim 7.3.9. *In the detouring graph, number of detouring cycles of size $\ell \geq 3$ is at most k^ℓ.*

Proof. Let $(u_1, \cdots, u_\ell)$ be an ℓ-tuple, where each entry is a terminal, that represents the terminal labels of a detouring cycle. By the Fact above, the ℓ-tuple determines uniquely all the vertices in the detouring cycle. By trivial counting, the number of possible ℓ-tuples is at most k^ℓ, and hence also the number of detouring cycles of size ℓ. □

Our key lemma is shows that for any $L \geq 3$, we can retain $\Omega_s(k^{L/(L-1)})$ vertices in the detouring graph, such that the induced graph on these vertices has *no* detouring cycle of size L or less.

Lemma 7.3.10. *For any integer $L \geq 3$, given a detouring graph with vertex set $\mathcal{K} = \{K_1, \cdots, K_r\}$, there exists a subset $\mathcal{K}' \subset \mathcal{K}$ of cardinality $\Omega_s(k^{L/(L-1)})$ such that the induced graph on $\mathcal{K}'$ has no detouring cycle of size L or less.*

Proof. We choose the subset $\mathcal{K}'$ by the following randomized algorithm:

1. Each vertex is picked into $\mathcal{K}'$ with probability $\delta k^{-(L-2)/(L-1)}$, where $\delta = \delta(s) < 1$ is a positive constant which we will derive explicitly later.

2. While (there is a detouring cycle of size L or less in the induced graph of $\mathcal{K}'$)
   Remove a vertex in the detouring cycle from $\mathcal{K}'$

After Step 1, $\mathbb{E}\left[|\mathcal{K}'|\right] = r \cdot \delta k^{-(L-2)/(L-1)} \geq \frac{\delta}{2s(s-1)} k^{L/(L-1)}$. Using Claim 7.3.9, the expected number of detouring cycles of size L or less is at most

$$\sum_{\ell=3}^{L} k^{\ell} \cdot (\delta k^{-(L-2)/(L-1)})^{\ell} \leq 2\delta^3 k^{L/(L-1)}.$$

Thus, the expected number of vertices removed in Step 2 is at most $2\delta^3 k^{L/(L-1)}$. Now, choose $\delta = 1/\sqrt{8s(s-1)}$. By the end of the algorithm,

$$\mathbb{E}\left[|\mathcal{K}'|\right] \geq \frac{\delta}{2s(s-1)} k^{L/(L-1)} - 2\delta^3 k^{L/(L-1)} = \Omega(k^{L/(L-1)}).$$

□

Construction of G and the proof. Recall the black-box reduction. Let k be an integer such that $K = [k]$ admits a $(9, 2)$-SS $\mathcal{K}$. By Lemma 7.3.10, we choose $\mathcal{K}'$ to be a subset of $\mathcal{K}$ with $|\mathcal{K}'| = \Omega(k^{5/4})$, such that the induced graph on $\mathcal{K}'$ has no detouring cycle of size 5 or less. We choose G^* to be a complete ternary tree of height 2, in which the 9 leaves are the terminals. For each $K_i \in \mathcal{K}'$, we add four non-terminals to K_i, altogether forming a *group*.

The following lemma is a direct consequence that the induced graph on $\mathcal{K}'$ has no detouring cycle of size 5 or less.

Lemma 7.3.11. *For any two terminals u, v in the same group, let R denote the group. Then, in any minor H of G, every simple path from u to v either comprises of R-edges only, or it comprises of edges from at least 5 groups other than R.*

Proof of Theorem 7.3.6. Let H be a $(2.5 - \epsilon)$-DAM of G, for some $\epsilon > 0$. Suppose that there exists a group such that all its non-terminals are not retained in H. By [119], there exists a pair of terminals u, v in that group such that every simple path between u and v, which comprises of edges of that group only, has length at least $3 \cdot \text{dist}_G(u, v)$.

By Lemma 7.3.11 and Lemma 7.3.4, any other simple path P between u and v passes through at least 4 other terminals, say they are u_a, u_b, u_c, u_d in the order of the direction from u to v. We denote this path by $P := u \to u_a \to u_b \to u_c \to u_d \to v$, by ignoring the non-terminals along the path. Between every pair of consecutive terminals in P, the length is at least 2. Thus, the length of P is at least 10. Since $\text{dist}_G(u, v) \leq 4$, the length of P is at least $2.5 \cdot \text{dist}_G(u, v)$.

Thus, the length of *every* simple path from u to v in H is at least $2.5 \cdot \text{dist}_G(u, v)$, a contradiction. Therefore, at least one non-terminal in each group is retained in H. As there are $\Omega(k^{5/4})$ groups, we are done. □

For the other results in Theorem 7.3.6, we follow the above proof almost exactly, with the following modifications. Set $s = 3^h$ for some $h \geq 2$, and set G^* to be a complete ternary tree with height h, in which the leaves are the terminals. Let α_h be a lower bound on the distortion for the SPR problem on G^*. Apply Lemma

7.3.10 with some integer $h < L \leq \lceil \alpha_h h \rceil$.[1] Following the above proof, attaining a distortion of $\min\left\{\frac{L}{h}, \alpha_h\right\} - \epsilon$ needs $\Omega(k^{L/(L-1)})$ non-terminals.

The last puzzle we need is the values of α_h. Chan et al. [58] considered unweighted complete binary tree with height h, and showed that as h tends to infinity, the minimum distortion of SPR problem tends to 8. However, it is not clear from their proof how the minimum distortion depends on h, which is needed for Theorem 7.3.6. In what follows, we use their ideas on unweighted complete ternary trees to derive such a dependence.

Let T_h denote a unweighted complete ternary tree of height h, where the leaves are the terminals. Let $\mathcal{S}_h$ denote the collection of all minors of T_h. For each of its node u, let $T(u)$ denote the sub-tree rooted at u, and let $t(u)$ denote the terminal which u contracts into. Denote the root by r, and its three children by x, y, z. Without loss of generality, we assume that r is contracted into a terminal t_r in $T(x)$, i.e., $t(r) = t_r$. Then, let[2]

$$\mathsf{DRL}(h, \alpha) := \min_{H \in \mathcal{S}_h,\, \text{distortion} \leq \alpha} \; \max_{\text{terminal } t \in T(y) \cup T(z)} \mathrm{dist}_H(t_r, t).$$

If there is not such a minor H, then $\mathsf{DRL}(h, \alpha) = +\infty$ by default. Note that when α increases, $\mathsf{DRL}(h, \alpha)$ decreases.

Let $H \in \mathcal{S}_h$ be a minor of T_h with distortion $\leq \alpha$. Let w denote a deepest node in $T(y) \cup T(z) \cup \{r\}$ such that $t(w) = t_r$. Let ℓ be the distance between r and w in T_h. Let w_1, w_2 be two children of w which are not in $T(x)$.

Then, by the definition of DRL, there exist two terminals $t_1 \in T(w_1)$ and $t_2 \in T(w_2)$ such that for $i = 1, 2$, $\mathrm{dist}_H(t_i, t(w_i)) \geq \mathsf{DRL}(h-\ell-1, \alpha)$. Also, for $i = 1, 2$, $\mathrm{dist}_H(t(w_i), t_r) \geq \mathrm{dist}_{T_h}(t(w_i), t_r) = 2h$. Hence,

$$\begin{aligned}\mathrm{dist}_H(t_1, t_2) &= \mathrm{dist}_H(t_1, t(w_1)) + \mathrm{dist}_H(t(w_1), t_r) + \mathrm{dist}_H(t_r, t(w_2)) \\ &\quad + \mathrm{dist}_H(t(w_2), t_2) \\ &\geq 2\left[\mathsf{DRL}(h - \ell - 1, \alpha) + 2h\right].\end{aligned}$$

Recall that $\mathrm{dist}_{T_h}(t_1, t_2) = 2(h - \ell)$. Hence, the distortion w.r.t. t_1, t_2 is at least

$$\frac{\mathsf{DRL}(h - \ell - 1, \alpha) + 2h}{h - \ell}.$$

This quantity cannot be larger than α.

We are ready to give a recurrence relation that bounds $\mathsf{DRL}(h, \alpha)$ from below:

$$\mathsf{DRL}(h, \alpha) \geq \min_{\ell \in [0, h-1]:\ \frac{\mathsf{DRL}(h-\ell-1, \alpha) + 2h}{h - \ell} \leq \alpha} \mathsf{DRL}(h - \ell - 1, \alpha) + 2h, \tag{7.1}$$

[1] Any choice of L larger than $\lceil \alpha_h h \rceil$ will not improve the result.

[2] Formally speaking, there can be infinitely many minors (with weights) of T_h with distortion at most α, so we should use $\inf$ instead of $\min$ in the definition. Yet, for each fixed minor without weight, the standard restriction [146, Definition 1.3] is the optimal weight assignment. Since there are only finitely many minors of T_h (without weights), we can replace $\inf$ by $\min$.

while the initial conditions are: $\forall \alpha \geq 1,\ \mathsf{DRL}(0,\alpha) = 0$, and

$$\mathsf{DRL}(1,\alpha) = \begin{cases} +\infty, & \text{if } \alpha < 2; \\ 2, & \text{if } \alpha \geq 2. \end{cases}$$

Let α_h denote the minimum distortion of T_h. By letting ℓ run over all possible distances between r and w, we obtain the following lower bound on α_h:

$$\alpha_h \geq \min_{\alpha} \ \max\left\{\alpha, \left(\min_{\ell \in [0,h-1]} \frac{\mathsf{DRL}(h-\ell-1,\alpha)+2h}{h-\ell}\right)\right\}. \tag{7.2}$$

We compute the lower bounds in (7.1) and (7.2) using math software. In the table below, we give the lower bounds on α_h for $h \in [3,10]$ and $h = 1000$.

h	2	3,4	5	6,7	8	9,10	1000
α_h	3	4	$22/5$	$14/3$	5	$26/5$	$257/35$

7.3.3 Generalizing the Lower Bound and its Implication

Indeed, we can set G^* as *any* graph. In our above proofs we used a tree for G^* because the only known lower bounds on distortion for the SPR problem are for trees. If one can find a graph G^* (either by a mathematical proof, or by computer searches) such that its distortion for the SPR problem is at least α, applying the black-box reduction with this G^*, and reusing the above proof show that there exists a graph G with k terminals such that attaining a distortion of $\alpha-\epsilon$ needs $\Omega(k^{1+\delta(G^*)})$ non-terminals, for some $\delta(G^*) > 0$.

Theorem 7.3.12. *Let G^* be a graph with s terminals, and the distance between any two terminals is between 1 and β. Suppose the distortion for the SPR problem on G^* is at least α. Then, for any positive integer $\max\{2, \lceil\beta\rceil\} \leq L \leq \lceil\alpha\beta\rceil$, there exists a constant $c_4 := c_4(s) > 0$, such that for infinitely many $k \in \mathbb{N}$, there exists a graph with k terminals which does not have a $\left(\min\{L/\beta, \alpha\} - \epsilon, c_4 k^{L/(L-1)}\right)$-DAM, for all $\epsilon > 0$.*

The above theorem has an interesting consequence. For the SPR problem on general graphs, the best known lower bound is 8, while the best known upper bound is $O(\log k)$ [102]. There is a huge gap between the two bounds, and it is not clear where the tight bound locates in between. Suppose that the tight lower bound on SPR is super-constant. Then for any positive constant α, there exists a graph G^*_α with $s(\alpha)$ terminals and some non-terminals, such that the distortion is larger than α. By Theorem 7.3.12, G^*_α can be used to construct a family of graphs with k terminals, such that to attain distortion α, the number of non-terminals needed is super-linear in k. Recall that in SPR, no non-terminal can be retained. In other words, Theorem 7.3.12 implies that: *if retaining no non-terminal will lead to a super-constant*

lower bound on distortion, then having the power of retaining any linear number of non-terminals will not improve the lower bound to a constant.

Formally, we define the following generalization of SPR problem. Let LSPR_y denote the problem that for an input graph with k terminals, find a DAM with at most yk non-terminals so as to minimize the distortion; the SPR problem is equivalent to LSPR_0.

Theorem 7.3.13. *For general graphs, SPR has super-constant lower bound on distortion if and only if for any constant $y \geq 0$, LSPR$_y$ has super-constant lower bound on distortion.*

7.4 Minor Construction for General Graphs

In this section we give minor constructions that present numerous trade-offs between the distortion and size of DAMs. Our results are obtained by combining the work of Coppersmith and Elkin [75] on sourcewise distance preservers with the well-known notion of spanners.

Given an undirected graph $G = (V, E, \mathbf{w})$, we let $\pi_{u,v}$ denote the shortest path between u and v in G. Without loss of generality, we assume that for any pair of vertices (u, v), the shortest path connecting u and v is *unique*. This can be achieved by slightly perturbing the original edge lengths of G such that no paths have exactly the same length (see [75]). The perturbation implies a *consistent* tie-breaking scheme: whenever π is chosen as the shortest path, every subpath of π is also chosen as the shortest path.

Let $N_G(u)$ denote the vertices incident to u in a graph G. We say that two paths π and π' branch at a vertex $u \in V(\pi) \cap V(\pi')$ iff $|N_{\pi \cup \pi'}(u)| > 2$. We call such a vertex u a *branching* vertex. Let $\mathcal{P}$ denote the set of shortest paths corresponding to every pair of vertices in G. We review the following result proved in [75, Lemma 7.5].

Lemma 7.4.1. *Any pair of shortest paths $\pi, \pi' \in \mathcal{P}$ has at most two branching vertices.*

To simplify our exposition, we introduce the notion of *terminal path covers*.

Definition 7.4.2 (Terminal Path Cover)**.** *Given $G = (V, E, \mathbf{w})$ with terminals K, a set of shortest paths $\mathcal{P}' \subset \mathcal{P}$ is an $(\alpha, f(k))$-terminal path cover (abbr. $(\alpha, f(k))$-TPc) of G with respect to K if*

1. *$\mathcal{P}'$ covers the terminals, i.e. $K \subseteq V(H)$, where $H = \bigcup_{\pi \in \mathcal{P}'} E(\pi)$,*
2. *$|\mathcal{P}'| \leq f(k)$ and for all $u, v \in K$, $dist_G(u, v) \leq dist_H(u, v) \leq \alpha \cdot dist_G(u, v)$.*

We remark that the endpoints of the shortest paths in $\mathcal{P}'$ are not necessarily terminals. The above definition naturally leads to the following algorithm, which is

Algorithm 7.1: MINORSPARSIFIER$(G, K, \mathcal{P}')$

Input : Graph $G = (V, E, \mathbf{w})$, terminals K, $(\alpha, f(k))$-TPc $\mathcal{P}'$ of G
Output : Distance Approximating Minor H of G

Set $H \leftarrow \emptyset$
Add all shortest paths from the path cover $\mathcal{P}'$ to H
while *there exists a degree two non-terminal* v *incident to edges* (v, u) *and* (v, w) **do**
    Contract the edge (u, v)
    Set the length of the edge (u, w) to $\text{dist}_G(u, w)$
return H

a slight generalization of the upper-bound technique employed by Krauthgamer et al. [170].

The following lemma gives an upper bound on the size of the DAM output by Algorithm 29. It is an easy generalization of a lemma in [170, Lemma 2.2] and we review it here for the sake completeness.

Lemma 7.4.3. *For a given graph $G = (V, E, \mathbf{w})$ with terminals $K \subset V$ and an $(\alpha, f(k))$-TPc $\mathcal{P}'$ of G, MINORSPARSIFIER(G,K,$\mathcal{P}'$) outputs an $(\alpha, f(k)^2)$-DAM of G.*

Proof. First, it is clear that the union over paths of $\mathcal{P}' \subset \mathcal{P}$ is a minor of G (this can be alternatively viewed as deleting non-terminals and edges that do not participate in any of the shortest paths in $\mathcal{P}'$). Further, the algorithm performs only edge contractions. Thus, the produced graph H is a minor of G.

Since contracting edges incident to non-terminals of degree two does not affect any distance in H, the distortion guarantee follows directly from that of the cover $\mathcal{P}'$. Thus, it only remains to show the bound on the size of H.

To this end, consider any two paths π, π' from $\mathcal{P}'$. From Lemma 7.4.1, we know that π and π' branch in at most two vertices. Let u_1 and u_2 denote such vertices. Due to the tie-breaking scheme in G, we know that the shortest path π_{u_1,u_2} is unique, and thus it must be shared by both π and π'. The latter implies that every vertex in the subpath must have degree degree exactly 2. Therefore, the only non-terminals in $\pi \cup \pi'$ are vertices u_1 and u_2, since non-terminals of degree two are removed from the edge contractions performed in the algorithm.

There are $O(f(k)^2)$ pairs of shortest paths from $\mathcal{P}'$, each having at most 2 non-terminals. Hence, the number of non-terminals in H is $O(f(k)^2)$. □

A trivial *exact* terminal path cover for any k-terminal graph is to take the union of all terminal shortest paths, which we refer to as the $(1, O(k^2))$-TPc $\mathcal{P}'$ of G. Krauthgamer et al. [170] used this $(1, O(k^2))$-TPc to construct an $(1, O(k^4))$-DAM. Here, we study the question of whether increasing the distortion slightly allows us to obtain a cover of size $o(k^2)$. We answer this question positively, by reducing it to the well-known spanner problem.

Let $q \geq 1$ be an integer and let $G = (V, E, \mathbf{w})$ be an undirected graph. A q-spanner of G is a subgraph $S = (V, E_S, \mathbf{w})$ such that for all u, v in V,

$\text{dist}_G(u,v) \leq \text{dist}_S(u,v) \leq q \cdot \text{dist}_G(u,v)$. We refer to q and $|E_S|$ as the *stretch* and *size* of spanner S, respectively. It is well-known that a simply greedy algorithm achieves the following guarantees.

Lemma 7.4.4 ([20]). *Let $q \geq 1$ be an integer. Any graph $G = (V, E, \mathbf{w})$ admits a $(2q-1)$-spanner S of size $O(|V|^{1+1/q})$.*

We use the above lemma as follows. Given a graph $G = (V, E, \mathbf{w})$ with terminals K, we compute the complete graph $Q_K = (K, \binom{K}{2}, d_G|K)$, where $d_G|K$ denotes the distance metric of G restricted to the point set K (In other words, for any pair of terminals $u, v \in K$, the weight of the edge connecting them in Q_K is given by $\mathbf{w}_{Q_K}(u,v) = d_G(u,v)$). Recall that all shortest paths in G are unique.

Using Lemma 7.4.4, we construct a $(2q-1)$-spanner S of size $O(k^{1+1/q})$ for Q_K. Observe that each edge of S corresponds to a unique (terminal) shortest path in G since S is a subgraph of Q_K. Thus, the set of shortest paths corresponding to edges of S form a $(2q-1, O(k^{1+1/q}))$-TPc $\mathcal{P}'$ of G. Using $\mathcal{P}'$ with Lemma 7.4.3 gives the following theorem.

Theorem 7.4.5. *Let $q \geq 1$ an integer. Any graph $G = (V, E, \mathbf{w})$ with $K \subset V$ admits a $(2q-1, O(k^{2+2/q}))$-DAM.*

We mention two trade-offs from the above theorem. When $q = 2$, we get an $(3, O(k^3))$-DAM. When $q = \log k$, we get an $(O(\log k), O(k^2))$-DAM. The above method allows us to have improved guarantees for bounded treewidth graphs. In particular, we prove that any graph G with treewidth at most p admits an $(O(\log p), O(pk))$-DAM.

Theorem 7.4.6. *Let $q \geq 1$ be an integer. Any graph $G = (V, E, \mathbf{w})$ with treewidth at most p, terminals $K \subset V$ and $k \geq p$ admits a $(2q-1, O(p^{1+2/q}k))$-DAM.*

Proof. We crucially exploit the fact that such graphs admit small separators: given a graph G of bounded treewidth p and any nonnegative vertex weight function $w(\cdot)$, there exists a set $S \subset V(G)$ of at most $p+1$ vertices whose removal separates the graph into two connected components, G_1 and G_2, each with $w(V(G_i)) \leq 2/3 w(V(G))$ (see [52]).

Krauthgamer et al. [170] use the above fact to construct an $(1, \mathcal{O}(p^3 k))$-DAM for graphs of treewidth at most p. We show that with two modifications, their algorithm can be extended to constructing distance approximating minors. The first modification is Step 2 of the algorithm ReduceGraphTW in [170]. For any integer $q \geq 1$, we replace their call to ReduceGraphNaive$(H, K \cup B)$[3] by our procedure MinorSparsifier$(H, K \cup B, \mathcal{P}')$, where $\mathcal{P}'$ is a $(2q-1, \mathcal{O}(p^{1+1/q}))$-TPc of G.

The second modification is a generalization of Lemma 4.2 in [170]. The main idea is to use the small separator set S to decompose the graph into smaller almost-disjoint graphs G_1 and G_2, compute their DAMs recursively, and then combine

[3]We remark that they use R to denote the set of terminals.

them using the separator S into a DAM of G. This implies that the separator S must belong to each G_i, i.e. all non-terminal vertices of S must be counted as additional terminals in each G_i. Below we give a formal definition of this decomposition/composition process.

Let $G_1 = (V_1, E_1, \mathbf{w}_1)$ and $G_2 = (V_2, E_2, \mathbf{w}_2)$ be graphs on disjoint sets of non-terminals, having terminal sets $K_1 = \{s_1, s_2, \ldots, s_{a_1}\}$ and $K_2 = \{t_1, t_2, \ldots, t_{a_2}\}$, respectively. Further, let $\phi(s_i) = t_i$, for all $i = 1, \ldots, c$ be an one-to-one correspondence between some subset of K_1 and K_2 (this correspondence is among the separator vertices). The *ϕ-merge* (or 2-sum) of G_1 and G_2 is the graph $G = (V, E, \mathbf{w})$ with terminal set $K = K_1 \cup \{t_{c+1}, \ldots, t_{a_2}\}$ formed by identifying the terminals s_i and t_i, for all $i = 1, \ldots, c$, where $\mathbf{w}(e) = \min\{\mathbf{w}_1(e), \mathbf{w}_2(e)\}$ (assuming infinite length when $\mathbf{w}_i(e)$ is undefined). We denote this operation by $G := G_1 \oplus_\phi G_2$.

Below we state the main lemma whose proof goes along the lines of [170, Lemma 4.2].

Lemma 7.4.7. *Let $G = G_1 \oplus_\phi G_2$. For $j = \{1, 2\}$, let H_j be an $(\alpha_j, f(a_j))$-DAM for G_j. Then the graph $H = H_1 \oplus_\phi H_2$ is an $(\max\{\alpha_1, \alpha_2\}, f(a_1) + f(a_2))$-DAM of G.*

In [170] it is shown that the size of the minor returned by the algorithm ReduceGraphTW is bounded by the number of leaves the in the recursion tree of the algorithm. Further, they prove that there are at most $O(k/p)$ such leaves. Plugging our bounds from the modification of Step 2 along with the above lemma yields our claimed result. □

7.5 Minor Construction for Fixed Minor-Free Graphs

In this section we give improved guarantees for distance approximating minors for special families of graphs. Specifically, we show that graphs that exclude a fixed minor admit an $(O(1), \tilde{O}(k^2))$-DAM. This family of graphs includes, among others, planar graphs.

The reduction to spanner in Section 7.4 does not consider the structure of Q_K, which is inherited from the input graph. We exploit this structure by employing the randomized Steiner Point Removal Problem, which is equivalent to finding an $(\alpha, 0)$-rDAM. Let us start by reviewing the following result of Englert et al. [94], which shows that for graphs that exclude a fixed minor, there exists a randomized minor with constant distortion.

Theorem 7.5.1 ([94], Theorem 14)**.** *Let $\alpha = O(1)$. Given a graph that excludes a fixed minor $G = (V, E, \mathbf{w})$ with $K \subset V$, there is a probability distribution η over minors $H = (K, E', \mathbf{w}')$ of G, such that for all u, v in K, $\mathbb{E}_{H\sim\eta}[dist_H(u, v)] \leq \alpha \cdot dist_G(u, v)$ and for every minor H in the support of η, $dist_H(u, v) \geq dist_G(u, v)$.*

Given a graph G that excludes a fixed minor, any minor H of G only on the terminals also excludes the same fixed minor. Thus H has $O(k)$ edges [244]. This leads to the corollary below.

Corollary 7.5.2. *Let $\alpha = O(1)$. Given a graph that excludes a fixed minor $G = (V, E, \mathbf{w})$ with $K \subset V$ and Q_K previously defined, there exists a probability distribution η over subgraphs $H = (K, E', \mathbf{w}')$ of Q_K, each having at most $O(k)$ edges, such that for all u, v in K, $\mathbb{E}_{H\sim\eta}[dist_H(u, v)] \leq \alpha \cdot dist_{Q_K}(u, v)$.*

Proof. Let η be the distribution over minors of G from Theorem 7.5.1, then every minor in its support is clearly a subgraph of Q_K with $O(k)$ edges. Since during the construction of these minors we may assume that for all (u, v) in E', $\mathbf{w}'(u, v) = \text{dist}_G(u, v)$, the corollary follows. □

Lemma 7.5.3. *Given a graph that excludes a fixed minor $G = (V, E, \mathbf{w})$ with $K \subset V$, and Q_K as previously defined, there exists an $O(1)$-spanner S of size $O(k \log k)$ for Q_K.*

Proof. Let η be the probability distribution over subgraphs H from Corollary 7.5.2. Set $S = \emptyset$. First, we sample independently $q = 3 \log k$ subgraphs $H_1, \ldots, H_q$ from η. We then add the edges from all these subgraphs to the graph S, i.e., $E_S = \bigcup_{i=1}^{q} E_{H_i}$. Fix an edge (t, t') from Q_K and a subgraph H_i. By Corollary 7.5.2 and the Markov inequality, $\mathbb{P}[\text{dist}_{H_i}(u, v) \geq 2\alpha \cdot \text{dist}_{Q_K}(u, v)] \leq 2^{-1}$, and hence

$$\mathbb{P}[\text{dist}_S(u, v) \geq 2\alpha \cdot d_{Q_K}(u, v)] = \prod_{i=1}^{q} \mathbb{P}[\text{dist}_{H_i}(u, v) \geq 2\alpha \cdot d_{Q_K}(t, t')] \leq 2^{-q} = k^{-3}.$$

Applying union bound overall all edges from Q_K yields

$$\mathbb{P}[\exists (u, v) \in E(Q_K) \text{ with } \text{dist}_S(u, v) \geq 2\alpha \cdot \text{dist}_{Q_K}(u, v)] \leq k^2 \cdot k^{-3} = k^{-1}.$$

Hence, for all edges (u, v) from Q_K, with probability at least $1 - 1/k$, we preserve the shortest path distance between u and v up to a factor of $2\alpha = O(1)$ in S. Since S is a subgraph of Q_K, this implies that there exists a $O(1)$-spanner S of size $O(k \log k)$ for Q_K. □

Similar to the last section, the set of shortest paths corresponding to edges of S is an $(O(1), O(k \log k))$-TPc $\mathcal{P}'$ of G. Using $\mathcal{P}'$ with Lemma 7.4.3 gives the following theorem.

Theorem 7.5.4. *Any graph that excludes a fixed minor $G = (V, E, \mathbf{w})$ with $K \subset V$ admits an $(O(1), \tilde{O}(k^2))$-DAM.*

7.6 Minor Construction for Planar Graphs

In this section, we show that for planar graphs one can improve the constant guarantee bound on the distortion to 3 and $1+\epsilon$, respectively, without affecting the size of the minor. Our work builds on existing techniques used in the context of approximate distance oracles, thereby bypassing our previous spanner reduction. Both results use essentially the same ideas and rely heavily on the fact that planar graphs admit separators with special properties.

We say that a graph $G = (V, E, \mathbf{w})$ admits a λ-separator if there exists a set $R \subseteq V$ whose removal partitions G into connected components, each of size at most λn, where $1/2 \leq \lambda < 1$. Lipton and Tarjan [184] showed that every planar graph has a $2/3$-separator R of size $O(\sqrt{n})$. Later on, Gupta et al. [121] and Thorup [245] independently observed that one can modify their construction to obtain a $2/3$-separator R, with the additional property that R consists of vertices belonging to shortest paths from G (note that this R is not guaranteed to be small). We briefly review the construction of such *shortest path separators*.

Let $G = (V, E, \mathbf{w})$ be a triangulated planar graph (the triangulation is guaranteed by adding infinity edge lengths among the missing edges). Further, let us fix an arbitrary shortest path tree A rooted at some vertex r. Then, it can be inferred from the work of Lipton and Tarjan [184] that there always exists a non-tree edge $e = (u, v)$ of A such that the fundamental cycle $\mathcal{C}$ in $A \cup \{e\}$, formed by adding the non-tree edge e to A, gives a $2/3$-separator for G. Because A is a tree, the separator will consist of two paths from the $\text{lca}(u, v)$ to u and v. We denote such paths by P_1 and P_2, respectively. Both paths are shortest paths as they belong to A. We will show how to use such separators to obtain terminal path covers. Before proceeding, we give the following preprocessing step.

Preprocessing Step. Given a planar graph $G = (V, E, \mathbf{w})$ with $K \subset V$, the algorithm MinorSparsifier(G, K, $\mathcal{P}'$) with $\mathcal{P}'$ being the $(1, O(k^2))$-TPc of G, produces an $(1, O(k^4))$-DAM G' for G. To simplify our notation, we will use G instead of G' in the following, i.e., we assume that G has at most $O(k^4)$ vertices.

7.6.1 Distortion-3 Guarantee

When solving a graph problem it is often the case that the solution is much easier on simpler graph instances, e.g., trees. Driven by this, it is desirable to reduce the problem from arbitrary graphs to one or several tree instances, possibly allowing a small loss in the quality of the solution. Along the lines of such an approach, Gupta et al. [121] gave the following definition in the context of shortest path distances.

Definition 7.6.1 (Forest Cover)**.** *Given a graph $G = (V, E, \mathbf{w})$, a forest cover (with stretch α) of G is a family $\mathcal{F}$ of subforests $\{F_1, F_2, \ldots, F_k\}$ of G such that for every $u, v \in V$, there is a forest $F_i \in \mathcal{F}$ such that $dist_G(u, v) \leq dist_{F_i}(u, v) \leq \alpha \cdot dist_G(u, v)$.*

If we restrict our attention to planar graphs, Gupta et al. [121] used shortest path separators (as described above) to give a divide-and-conquer algorithm for constructing forest covers with small guarantees on the stretch and size. Here, we slightly modify and adopt their construction for our specific application. Before proceeding to the algorithm, we give the following useful definition.

Definition 7.6.2. *Let t be a terminal and let π be a shortest path in G. Then $t^{\pi}_{\min}$ denotes the vertex of π that minimizes $dist_G(t, p)$, for all $p \in V(\pi)$, breaking ties lexicographically.*

Algorithm 7.2: ForestCover(G, K)

Input : Planar graph $G = (V, E, \mathbf{w})$, terminals K
Output : Forest cover $\mathcal{F}$ of G

if $|V(G)| \leq 1$ **then**
  return $V(G)$
Compute a $2/3$-separator $\mathcal{C}$ consisting of shortest paths π_1 and π_2 for G
for $i = 1, 2$ **do**
  Contract π_i to a single vertex p_i and compute a shortest path tree L_i from p_i
  Expand back the contracted edges in L_i to get the tree L'_i
  for *every terminal* $t \in K$ **do**
    Add $t^{\pi_i}_{\min}$ as a terminal in the tree L'_i
Let (G_1, K_1) and (G_2, K_2) be the resulting connected graphs from $G \setminus \mathcal{C}$, where K_1 and K_2 are disjoint subsets of the terminals K induced by $\mathcal{C}$
`// Note that all distances involving terminals from C are taken care of`
return $\bigcup_{i=1}^{2} L'_i \cup \bigcup_{i=1}^{2}$ ForestCover(G_i, K_i)

Algorithm 7.3: PlanarTPc-1 (G, K)

Input : Planar graph $G = (V, E, \mathbf{w})$, terminals K
Output : Terminal path cover $\mathcal{P}'$ of G

Set $\mathcal{P}' \leftarrow \emptyset$
Set $\mathcal{F} \leftarrow$ ForestCover(G, K)
for *each forest* $F_i \in \mathcal{F}$ **do**
  Let R_i be the terminal set of F_i and let $\mathcal{P}'_i$ be the (trivial) $(1, O(k^2))$-TPc of F_i
  Compute F'_i *gets* MinorSparsifier$(F_i, R_i, \mathcal{P}'_i)$
  Add the shortest paths corresponding to the edges of F'_i to $\mathcal{P}'$
return $\mathcal{P}'$

Gupta et al. [121] showed the following guarantees for Algorithm 30.

Theorem 7.6.3 ([121], Theorem 5.1)**.** *Given a planar graph $G = (V, E, \mathbf{w})$ with $K \subset V$, ForsetCover(G, K) produces a stretch-3 forest cover with $O(\log |V|)$ forests.*

We note that the original construction does not consider terminal vertices, but this does not worsen neither the stretch nor the size of the cover. The only difference here is that we need to add at most k new terminals to each forest compared to the original number of terminals in the input graph. This modification affects our bounds on the size of a minor only by a constant factor.

Below we show that using the above theorem one can obtain terminal path covers for planar graphs.

Lemma 7.6.4. *Given a planar graph $G = (V, E, \mathbf{w})$ with $K \subset V$, PlanarTPc-1(G, K) produces an $(3, O(k \log k))$-TPc $\mathcal{P}'$ for G.*

Proof. We first review the following simple fact, whose proof can be found in [170].

Fact 7.6.5. *Given a forest $F = (V, E, \mathbf{w})$ with terminals $K \subset V$ and $\mathcal{P}'$ being the (trivial) $(1, O(k^2))$-TPc of F, the procedure MinorSparsifier$(F, K, \mathcal{P}')$ outputs an $(1, k)$-DAM.*

Let us proceed with the analysis. Observe that from the Preprocessing Step our input graph G has at most $O(k^4)$ vertices. Thus, applying Theorem 7.6.3 on G gives a stretch-3 forest cover $\mathcal{F}$ of size $O(\log k)$. In addition, recall that all shortest paths are unique in G.

Next, let F_i by any forest from $\mathcal{F}$. By construction, we note that each tree belonging to F_i has the nice property of being a concatenation of a given shortest path with another shortest path tree. We will exploit this in order to show that every edge of the minor F_i' for F_i corresponds to the (unique) shortest path between its endpoints in G.

To this end, let $e' = (u, v)$ be an edge of F_i' that does not exist in F_i. Since F_i' is a minor of F_i, we can map back e' to the path $\pi_{u,v}$ connecting u and v in F_i. Because of the additional terminals $u_{\min}^{P_i}$ added to F_i, we claim that $\pi_{u,v}$ is entirely contained either in some shortest path tree L_j or some shortest path separator P_j. Using the fact that subpaths of shortest paths are shortest paths, we conclude that the length of the path $\pi_{u,v}$ (or equivalently, the length of edge e') corresponds to the unique shortest path connecting u and v in G. The same argument is repeatedly applied to every such edge of F_i'.

By construction we know that F_i has at most $2k$ terminals. Using Fact 7.6.5 we get that F_i' contains at most $4k$ edges. Since there are $O(\log k)$ forests, we conclude that the terminal path cover $\mathcal{P}'$ consists of $O(k \log k)$ shortest paths. The stretch guarantee follows directly from that of cover $\mathcal{F}$, since F_i' exactly preserves all distances between terminals in F_i. □

Theorem 7.6.6. *Any planar graph $G = (V, E, \mathbf{w})$ with $K \subset V$ admits a $(3, \tilde{O}(k^2))$-DAM.*

7.6.2 Distortion-$(1+\epsilon)$ Guarantee

Next we present our best trade-off between distortion and size of minors for planar graphs. Our idea is to construct terminal path covers using the construction of Thorup [245] in the context of approximate distance oracles in planar graphs. Here, we modify a simplified version due to Kawarabayashi et al. [155].

The construction relies on two important ideas. Similarly to the distortion-3 result, the first idea is to recursively use shortest path separators to decompose the graph. The second consists of approximating shortest paths that cross a shortest path separator. Below we present some necessary modification to make use of such a construction for our purposes.

Let π be a shortest path in G. For a terminal $t \in K$, we let the pair (p,t), where $p \in V(\pi)$, denote the *portal* of t with respect to the path π. An ϵ-cover $C(t,\pi)$ of t with respect to π is a set of portals with the following property:

- for all $p \in V(\pi)$, thee exsits $q \in C(t,\pi)$ such that

$$\text{dist}_G(t,q) + \text{dist}_G(q,p) \leq (1+\epsilon)\text{dist}_G(t,p).$$

Let (t,t') by any terminal pair in G. Let $\pi_{t,t'}$ be the (unique) shortest path that crosses the path π at vertex w. Then using the ϵ-covers $C(t,\pi)$ and $C(t',\pi)$, there exist portals (t,p) and (p',t') such that the new distance between t and t' is

$$\begin{aligned}\text{dist}_G(t,p) + \text{dist}_G(p,p') + \text{dist}_G(p',t') \\ &\leq \text{dist}_G(t,p) + \text{dist}_G(p,w) + \text{dist}_G(w,p') + \text{dist}_G(p',t') \\ &\leq (1+\epsilon)\text{dist}_G(t,t').\end{aligned} \tag{7.3}$$

The new distance clearly dominates the old one. The next result due to Thorup [245] shows that maintaining a small number of portals per terminal suffices to approximately preserve terminal shortest paths.

Lemma 7.6.7. *Let $\epsilon > 0$. For a given terminal $t \in K$ and a shortest path π, there exists an ϵ-cover $C(t,\pi)$ of size $O(1/\epsilon)$.*

The above lemma leads to the following recursive procedure.

Lemma 7.6.8. *Given a planar graph $G = (V,E,\mathbf{w})$ with $K \subset V$, PlanarTPc-2(G,K) outputs an $(1+\epsilon, O(k\epsilon^{-1}\log k))$-TPc $\mathcal{P}'$ for G.*

Proof. From the Preprocessing Step we know that G has at most $O(k^4)$ vertices. Further, recall that removing the vertices that belong to the shortest path separators from G results into two graphs G_1 and G_2, whose size is at most $2/3 \cdot |G|$. Thus, there are at most $O(\log k)$ levels of recursion for the above procedure.

Let $\mathcal{P}'$ be the terminal path cover output by PlanarTPc-2(G,K). We first bound the number of separator shortest paths added in Step 3. Note that at any level of the recursion there at most k terminals and, thus the number of recursive calls per

Algorithm 7.4: PLANARTPC-2 (G, K)

Input : Planar graph $G = (V, E, \mathbf{w})$, terminals K
Output : Terminal path cover $\mathcal{P}'$ of G

if $|V(G)| \leq 1$ *or* $K = \emptyset$ **then**
  return $\emptyset$
Set $\mathcal{B} \leftarrow \emptyset$
Compute a $2/3$-separator $\mathcal{C}$ consisting of shortest paths π_1 and π_2
Add π_1 and π_2 to $\mathcal{B}$
for *every terminal* $t \in K$ **do**
  Compute ϵ-covers $C(t, \pi_1)$ and $C(t, \pi_2)$
  for *every portal* $(t, p) \in C(t, \pi_1) \cup C(t, \pi_2)$ **do**
    Add the shortest path $\pi_{t,p}$ to $\mathcal{B}$
Let (G_1, K_1) and (G_2, K_2) be the resulting connected graphs from $G \setminus \mathcal{C}$, where K_1 and K_2 are disjoint subsets of the terminals K induced by $\mathcal{C}$
// Note that all distances involving terminals from $\mathcal{C}$ are taken care of
return $\mathcal{B} \cup \bigcup_{i=1}^{2}$ PLANARTPC-2(G_i, K_i)

level is at most k. Since we added two paths per recursive call, we get that there are at most $O(k \log k)$ paths overall.

We now continue with the counting or portals. Let $t \in K$ be any terminal and consider any recursive call applied on the current graph (G', K'). If $t \notin K'$, then we simply ignore t. Otherwise, t either belongs to one of the separator shortest paths in G' or one of the partitions induced by the separators. In the first case, we know that t is retained because we added π_1 and π_2 to $\mathcal{P}'$ and these are already counted. In the second case, using Lemma 7.6.7, we add $O(1/\epsilon)$ shortest paths connecting portals from $C(t, \pi_1)$ and $C(t, \pi_2)$. Therefore, in any recursive call, we maintain at most $O(1/\epsilon)$ shortest paths per terminal. Since every terminal can participate in at most $O(\log k)$ recursive calls, we get that the total number of portal-shortest paths is at most $O(k \log k/\epsilon)$. Combining both bounds, it follows that the size of $\mathcal{P}'$ is at most $O(k \log k/\epsilon)$.

It remains to show the stretch guarantee of $\mathcal{P}'$. Let R be the recursion tree of the algorithm, where every node corresponds to a recursive call. For any pair $t, t' \in L$, let $a \in V(R)$ associated with (G_a, K_a) be the leafmost node such that $t, t' \in K_a$. Then, it follows that among all ancestors of a in the tree R, there must exist a separator path π_i, $i = 1, 2$ that crosses $\pi_{t,t'}$ and attains the minimum length. The stretch guarantee follows directly from (7.3). □

Theorem 7.6.9. *Any planar graph $G = (V, E, \mathbf{w})$ with $K \subset V$ admits an $(1 + \epsilon, O(k^2\epsilon^{-2} \log^2 k))$-DAM.*

7.7 Conclusion

In this chapter, we introduced the notion of distance approximating minors, which are vertex sparsifiers that are minors of the input graph and approximately preserve shortest path distances among a designated subset of vertices, referred to as terminals. This notion is a natural generalization of the Steiner Point Removal problem [119], where the sparsifier must contain only terminals and the Distance Preserving Minor problem [170], where we want to exactly preserve pair-wise terminal distances while allowing additional non-terminal vertices in the sparsifier. We study distance approximating minors from both upper and lower bound perspective. For example, we show that for k-terminal general graphs and distortion $3-\epsilon$, one needs to retain at least $\Omega(k^{6/5})$ non-terminal vertices. For planar graphs, we show an algorithm that computes a $(1+\epsilon)$-distance approximating minors with $\tilde{O}(k^2\epsilon^{-2})$ non-terminals. Our lower-bound and algorithmic constructions bring together techniques from distance oracles, branching events in shortest path computations and Steiner systems from combinatorics.

There remain gaps between some of the best upper and lower bounds, e.g., for distortion $3-\epsilon$, the lower bound is $\Omega(k^{6/5})$, while for distortion 3, our upper bound is $\tilde{O}(k^3)$. Therefore, understanding the trade-off between distortion and the size of the sparsifiers is an interesting open problem. In the same vein, it is interesting to explore whether the size of the sparisifer in the planar setting can be improved to $\tilde{O}(k^{2-o(1)})$, while keeping the same approximation guarantee. As we demonstrate in Chapter 6, this question is particularly relevant due to its connection to the offline dynamic APSP problem in planar graphs.

Another important problem in this area is to design fast algorithms for constructing distance preserving minors. While most of the vertex sparsification studies in the literature have focused on understanding the trade-off between distortion and size, we believe that the running time for constructing such sparsifiers is an important aspect that better serves the general purpose of using sparsification to speed up algorithmic constructions.

CHAPTER 8

Reachability Preserving Minors and Sparsifiers for Cuts and Distances

Graph Sparsification aims at compressing large graphs into smaller ones while preserving important characteristics of the input graph. In this chapter we study Vertex Sparsifiers, i.e., sparsifiers whose goal is to reduce the number of vertices. We focus on the following notions:

(1) Given a digraph $G = (V, E)$ and terminal vertices $K \subset V$ with $|K| = k$, a (vertex) reachability sparsifier of G is a digraph $H = (V', E')$, $K \subset V'$ that preserves all reachability information among terminal pairs. In this chapter we introduce the notion of reachability-preserving minors (RPMs) , i.e., we require H to be a minor of G. We show any directed graph G admits a RPM H of size $O(k^3)$, and if G is planar, then the size of H improves to $O(k^2 \log k)$. We complement our upper-bound by showing that there exists an infinite family of grids such that any RPM must have $\Omega(k^2)$ vertices.

(2) Given a weighted undirected graph $G = (V, E)$ and terminal vertices K with $|K| = k$, an exact (vertex) cut sparsifier of G is a graph H with $K \subset V'$ that preserves the value of minimum-cuts separating any bipartition of K. We show that planar graphs with all the k terminals lying on the same face admit exact cut sparsifiers of size $O(k^2)$ that are also planar. Our result extends to flow and distance sparsifiers. It improves the previous best-known bound of $O(k^2 2^{2k})$ for cut and flow sparsifiers by an exponential factor, and matches an $\Omega(k^2)$ lower-bound for this class of graphs.

8.1 Introduction

Very large graphs or networks are ubiquitous nowadays, from social networks to information networks. One natural and effective way of processing and analyzing such graphs is to compress or sparsify the graph into a smaller one that well preserves certain properties of the original graph. Such a sparsification can be obtained by reducing the number of *edges*. Typical examples include cut sparsifiers [35], spectral sparsifiers [238], spanners [251] and transitive reductions [15], which are subgraphs defined on the same vertex set of the original graph G while having much smaller number of edges and still well preserving the cut structure, spectral properties, pairwise distances and transitive closure of G, respectively.

Another way of performing sparsification is by reducing the number of *vertices*, which is most appealing when only the properties among a subset of vertices (which are called *terminals*) are of interest (see e.g., [24, 170, 197]). We call such small graphs *vertex sparsifiers* of the original graph. In this chapter, we will particularly focus on vertex reachability sparsifiers for *directed* graphs and cut (and other related) sparsifiers for *undirected* graphs.

Vertex reachability sparsifiers in directed graphs is an important and fundamental notion in Graph Sparsification, which has been implicitly studied in the dynamic graph algorithms community [81, 240], and explicitly in [154]. Specifically, given a digraph $G = (V, E)$, $K \subset V$, a digraph $H = (V', E')$, $K \subset V'$ is a (*vertex*) *reachability sparsifier* of G if for any $x, x' \in K$, there is a directed path from x to x' in H iff there is a directed path from x to x' in G. If $|K| = k$, we call the digraph G a *k-terminal digraph*. Note that any k-terminal digraph G always admits a trivial reachability vertex sparsifier H, which corresponds to the transitive closure restricted to the terminals.

In this chapter, we initiate the study of *reachability-preserving minors*, i.e., vertex reachability sparsifiers with H required to be a minor of G. The restriction on H being a minor of G is desirable as it makes sure that H is structurally similar to G, e.g., any minor of a planar graph remains planar. We ask the question whether general graphs admit reachability-preserving minors whose size can be bounded independently of the input graph G, and study it from both the lower- and upper-bound perspective.

For the notion of cut (and other related) sparsifiers, we are given a capacitated undirected graph $G = (V, E, \mathbf{c})$, and a set of terminals K and our goal is to find a (capacitated undirected) graph $H = (V', E', \mathbf{c}')$ with as few vertices as possible and $K \subseteq V'$ such that the quantities like, cut value, multi-commodity flow and distance among terminal vertices in H are the same as or close to the corresponding quantities in G. If $|K| = k$, we call the graph G a *k-terminal graph*.

We say H is a *quality-q* (*vertex*) *cut sparsifier* of G, if for every bipartition $(U, K \setminus U)$ of the terminal set K, the value of the minimum cut separating U from $K \setminus U$ in G is within a factor of q of the value of minimum cut separating U from $K \setminus U$ in H. If H is a quality-1 cut sparsifier, then it will be also called a *mimicking network* [125]. Similarly, we define flow and distance sparsifiers that (ap-

proximately) preserve multicommodity flows and distances among terminal pairs, respectively (see Section 8.6 for formal definitions). These type of sparsifiers have proven useful in approximation algorithms [197] and also find applications in network routing [73].

Our Results. Our first and main contribution is the study of reachability-preserving minors. Although reachability is a weaker requirement in comparison to shortest path distances, directed graphs are usually much more cumbersome to deal with from the perspective of graph sparsification. Surprisingly, we show that general digraphs admit reachability-preserving minors with $O(k^3)$ vertices, which is in contrast to the bound of $O(k^4)$ on the size of distance-preserving minors in undirected graphs by Krauthgamer et al. [170].

Theorem 8.1.1. *Given a k-terminal digraph G, there exist a reachability-preserving minor H of G with size $O(k^3)$.*

It might be interesting to compare the above result with the construction of *reachability preserver* by Abbound and Bodwin [1], where the reachability preserver for a pair-set P in a graph G is defined to be a *subgraph* of G that preserves the reachability of all pairs in P. The size (i.e., the number of edges) of such preservers is shown to be at least $\Omega(n^{2/(d+1)}|P|^{(d-1)/d})$, for any integer $d \geq 2$, which is in sharp contrast to our upper bound $O(|P|^{3/2})$ on the size of reachability-preserving minors by taking P to be the pair-set of all terminals.

Furthermore, by exploiting a tight integration of our techniques with the compact distance oracles for planar graphs by Thorup [245], we can show the following theorem regarding the size of reachability-preserving minors for planar digraphs.

Theorem 8.1.2. *Given a k-terminal planar digraph G, there exists a reachability-preserving minor H of G with size $O(k^2 \log k)$.*

We complement the above result by showing that there exist instances where the above upper-bound is tight up to a $O(\log k)$ factor.

Theorem 8.1.3. *For infinitely many $k \in \mathbb{N}$ there exists a k-terminal acyclic directed grid G such that any reachability-preserving minor of G must use $\Omega(k^2)$ non-terminals.*

Our second contribution is new algorithms for constructing quality-1 (exact) cut, flow and distance sparsifiers for k-terminal planar graphs, where all the terminals are assumed to lie on the same face. We call such k-terminal planar graphs *Okamura-Seymour* (OS) instances. They are of particular interest in the algorithm design and optimization community, due to the classical Okamura-Seymour theorem that characterizes the existence of feasible concurrent flows in such graphs (see e.g., [64, 65, 179, 205]).

We show that the size of quality-1 sparsifiers can be as small as $O(k^2)$ for such instances, for which only exponential (in k) size of cut and flow sparisifiers were known before [24, 171]. Formally, we have the following theorem.

Theorem 8.1.4. *For any OS instance G, i.e., a k-terminal planar graph in which all terminals lie on the same face, there exist quality-1 cut, flow and distance sparsifers of size $O(k^2)$. Furthermore, the resulting sparsifiers are also planar.*

We remark that all the above sparsifiers can be constructed in polynomial time (in n and k), but we will not optimize the running time here. As we mentioned above, previously the only known upper bound on the size of quality-1 cut and flow sparsifiers for OS instance was $O(k^2 2^{2k})$, given by [24, 171]. Our upper bound for cut sparsifier also matches the lower bound of $\Omega(k^2)$ for OS instance given by [171]. More specifically, in [171], an OS instance (that is a grid in which all terminals lie on the boundary) is constructed, and used to show that any mimicking network for this instance needs $\Omega(k^2)$ edges, which is thus a lower bound for planar graphs (see the table below for an overview). Note that that even though our distance sparsifier is not necessarily a minor of the original graph G, it still shares the nice property of being planar as G. Furthermore, Krauthgamer and Zondiner [173] proved that there exists a k-terminal planar graph G (not necessarily an OS instance), such that any quality-1 distance sparsifier of G that is planar requires at least $\Omega(k^2)$ vertices.

Graph	Type of sparsifier	Upper Bound	Lower Bound
Planar	Cut (minor)	$O(k2^{2k})$ [171]	$\Omega(k^2)$ [171]
Planar (γ)	Cut (minor)	$O(\gamma^5 2^{2\gamma} k^4)$ [172]	$\Omega(2^k)$ [153]
Planar OS	Cut (planar)	$\mathbf{O(k^2)}$	$\Omega(k^2)$ [171]
Planar OS	Distance (minor)	$O(k^4)$ [170]	$\Omega(k^2)$ [170]
Planar OS	Distnace (planar)	$\mathbf{O(k^2)}$	$\Omega(k^2)$ [173]

Table 8.1: An overview on the best-known results for mimicking networks and distance sparsifiers. The results which are *not* followed by a reference are shown in this chapter.

We further provide a lower bound on the size of any *data structure* (not necessarily a graph) that approximately preserves pairwise terminal distances of *general* k-terminal graphs, which gives a trade-off between the distance stretch and the space complexity.

Theorem 8.1.5. *For any $\varepsilon > 0$ and $t \geq 2$, there exists a (sparse) k-terminal n-vertex graph such that $k = o(n)$, and any data structure that approximates pairwise terminal distances within a multiplicative factor of $t - \varepsilon$ or an additive error $2t - 3$ must use $\Omega(k^{1+1/(t-1)})$ bits.*

Remark. Recently and independently of our work, Krauthgamer and Rika [172] constructed quality-1 cut sparsifiers of size $O(\gamma^5 2^{2\gamma} k^4)$ for planar graphs whose terminals are incident to at most $\gamma = \gamma(G)$ faces. In comparison with our upper-bound which only considers the case $\gamma = 1$, the size of our sparsifiers from Theorem 8.1.4 is better by a $\Omega(k^2)$ factor. Moreover, their work focuses on constructing sparsifiers

that are minors of the originial input graph, while our construction only guarantee that the resulting sparsifiers are planar graphs. Subsequent to our work, Karpov et al. [153] proved that there exists edge-weighted k-terminal planar graphs that require $\Omega(2^k)$ edges in any exact cut sparsifier, which implies that it is necessary to have some additional assumption (e.g., $\gamma = O(1)$) to obtain a cut sparsifier of $k^{O(1)}$ size.

Our Techniques. Our results for reachability-preserving minors are obtained by exploiting a technique of counting "branching" events between shortest paths in the directed setting (this technique was introduced by Coppersmith and Elkin [75], and has also been recently leveraged by Bodwin [53] and Abboud and Bodwin [1]). Using this and a consistent tie-breaking scheme for shortest paths, we can efficiently construct a RPM for general digraphs of size $O(k^4)$ and by using a more refined analysis of branching events (see [1]), we can further reduce the size to be $O(k^3)$. We then combine our construction with a decomposition for planar digraphs (see [245]), to show that it suffices to maintain the reachability information among $O(k \log k)$ terminal pairs, instead of the naive $O(k^2)$ pairs, and then construct a RPM for planar digraphs with $O(k^2 \log k)$ vertices.

The lower-bound follows by constructing a special class of k-terminal directed grids and showing that any RPM for such grids must use $\Omega(k^2)$ vertices. Similar ideas for proving the lower bound on the size of distance-preserving minors for undirected graphs have been used by Krauthgamer et al. [170].

We construct our quality-1 cut and distance sparsifiers by repeatedly performing *Wye-Delta transformations*, which are local operations that preserve cut values and distances and have proven very powerful in analyzing electrical networks and in the theory of circular planar graphs (see e.g., [77, 101]). Khan and Raghavendra [162] used Wye-Delta transformations to construct quality-1 cut sparsifiers of size $O(k)$ for trees and outerplanar graphs, while our case (i.e., the planar OS instances) is more general and complicated and previously it was not clear at all how to apply such transformations to a more broad class of graphs. Our approach is as follows. Given a k-terminal planar graph with terminals lying on the same face, we first embed it into some large grid with terminals lying on the boundary of the grid. Next, we show how to embed this grid into a "more suitable" graph, which we will refer to as "half-grid". Finally, using the Wye-Delta operations, we reduce the "half-grid" into another graph whose number of vertices can be bounded by $O(k^2)$. Since we argue that the above graph reductions preserve exactly all terminal minimum cuts, our result follows. Gitler [112] proposed a similar approach for studying the reducibility of multi-terminal graphs with the goal to classify all Wye-Delta reducible graphs, which is very different from our motivation of constructing small vertex sparsifiers with good quality.

The distance sparsifiers can be constructed similarly by slightly modifying the Wye-Delta operation. Our flow sparsifiers follow from the construction of cut sparsifiers and the flow/cut gaps for OS instances (which has been initially observed by

Andoni et al. [24]). Our lower bound on the space complexity of any compression function approximately preserving terminal pairwise distance is derived by combining extremal combinatorics construction of Steiner Triple System that was used to prove lower bounds on the size of distance approximating minors (see [69]) and the incompressibility technique from [192].

Related Work. There has been a long line of work on investigating the tradeoff between the quality of the vertex sparsifier and its size (see e.g., [24, 94, 171]). (Throughout, cut, flow and distance sparsifiers will refer to their vertex versions.) Quality-1 *cut sparsifiers* (or equivalently, mimicking networks) were first introduced by Hagerup et al. [125], who proved that for any graph G, there always exists a mimicking network of size $O(2^{2^k})$. Krauthgamer and Rika [171] showed how to build a mimicking network of size $O(k^2 2^{2k})$ for any planar graph G that is minor of the input graph. They also proved a lower bound of $\Omega(k^2)$ on the number of edges of the mimicking network of planar graphs, and a lower bound of $2^{\Omega(k)}$ on the number of vertices of the mimicking network for general graphs.

Quality-1 vertex flow sparsifiers have been studied in [24, 118], albeit only for restricted families of graphs like quasi-bipartite, series-parallel, etc. It is not known if any general undirected graph G admits a constant quality flow sparsifier with size independent of $|V(G)|$ and the edge capacities. For the quality-1 distance sparsifiers, Krauthgamer et al. [170] introduced the notion of *distance-preserving minors*, and showed an upper-bound of size $O(k^4)$ for general undirected graphs. They also gave a lower bound of $\Omega(k^2)$ on the size of such a minor for planar graphs. Abboud et al. [5] show how to compress a planar graph metric using only $\tilde{O}(\min\{k^2, \sqrt{k \cdot n}\})$ bits. Recently, Chang et al. [59] extended their compressing scheme to a graph sparsifer which matches their bound.

Over the last two decades, there has been a considerable amount of work on understanding the tradeoff between the sparsifier's quality q and its size for $q > 1$, i.e., when the sparsifiers only *approximately* preserve the corresponding properties [24, 44, 58, 60, 63, 68, 69, 72, 94, 102, 109, 119, 146, 180, 191, 197].

8.2 Preliminaries

Let $G = (V, E)$ be a directed graph with terminal set $K \subset V$, $|K| = k$, which we will refer to as a *k-terminal digraph*. We say G is a *k-terminal* DAG if G has no directed cycles. The *in-degree* of a vertex v, denoted by $\deg^-_G(v)$, is the number of edges directed towards v in G. A digraph $H = (V', E')$, $K \subset V'$ is a (*vertex*) *reachability sparsifier* of G if for any $x, x' \in K$, there is a directed path from x to x' in H iff there is a directed path from x to x' in G. If H is obtained by performing minor operations in G, then we say that H is a *reachability-preserving minor* of G. We define the *size* of H to be the number of non-terminals in H, i.e. $|V' \setminus K|$.

Let $G = (V, E, \mathbf{c})$ be an undirected graph with terminal set $K \subset V$ of cardinality k, where $\mathbf{c} : E \to \mathbb{R}_{\geq 0}$ assigns a non-negative capacity to each edge. We will refer to such a graph as a *k-terminal graph*. Let $U \subset V$ and $S \subset K$. We say

that a cut $(U, V \setminus U)$ is S-separating if it separates the terminal subset S from its complement $K \setminus S$, i.e., $U \cap K$ is either S or $K \setminus S$. We will refer to such cut as a *terminal cut*. The cutset $\delta(U)$ of a cut $(U, V \setminus U)$ represents the edges that have one endpoint in U and the other one in $V \setminus U$. The cost $\mathrm{cap}_G(\delta(U))$ of a cut $(U, V \setminus U)$ is the sum over all capacities of the edges belonging to the cutset. We let $\text{mincut}_G(S, K \setminus S)$ denote the minimum cost of any S-separating cut of G. A graph $H = (V', E', \mathbf{c}')$, $K \subset V'$ is a *quality-q (vertex) cut sparsifier* of G with $q \geq 1$ if for any $S \subset K$, $\text{min-cut}_G(S, K \setminus S) \leq \text{min-cut}_H(S, K \setminus S) \leq q \cdot \text{min-cut}_G(S, K \setminus S)$.

8.3 Reachability-Preserving Minors for General Digraphs

In this section, we provide two constructions for reachability-preserving minors for general digraphs. The resulting minor from the first construction has size $O(k^4)$, which is larger than the size $O(k^3)$ of the minor from the second construction. However, our first construction can be implemented in polynomial time (in n), while the second one requires exponential running time.

8.3.1 A Warm-up: An Upper Bound of $O(k^4)$

In this section we show that any k-terminal digraph admits a reachability-preserving minor of size $O(k^4)$. We accomplish this by first restricting our attention to DAGs, and then showing how to generalize the result to any digraph.

We start by introducing the following definition. Given a digraph G with a terminal set K of size k and a pair-set $P \subseteq K \times K$, we say that H is a reachability-preserving minor with respect to P, if H is a minor of G that preserves the reachability information only among the pairs in P. Note that in the definition of vertex reachability sparsifiers, the *trivial* pair-set P contains $k(k-1)$ terminal-pairs, i.e., for any pair $x, x' \in K$, both (x, x') and (x', x) belong to P. Whenever we omit P, we mean to preserve the reachability information among all possible terminal pairs.

We next review a useful scheme for breaking ties between shortest paths connecting some vertex pair from P. This tie-breaking is usually achieved by slightly perturbing the edge lengths of the original graph such that no two paths have the same length (note that in our case, edge lengths are initially one). The perturbation gives a *consistent* scheme in the sense that whenever π is chosen as a shortest path, every sub-path of π is also chosen as a shortest path. Below we formalize these ideas using two definitions and a lemma from [53].

Definition 8.3.1 (Tie-breaking Scheme)**.** *Given a k-terminal G, a shortest path tie breaking scheme is a function π that maps every pair of vertices (s, t) to some shortest path between s and t in G. For any pair-set P, we let $\pi(P)$ denote the union over all shortest paths between pairs in P with respect to the scheme π.*

Definition 8.3.2 (Consistency)**.** *A tie-breaking scheme is consistent if, for all vertices $y, x, x', y' \in V$, if $x, x' \in \pi(y, y')$ with $d(y, x) < d(y, x')$, then $\pi(x, x')$ is a sub-path of $\pi(y, y')$.*

Lemma 8.3.3 ([53])**.** *For any k-terminal G, there is a consistent tie-breaking scheme in G.*

We remark that for any k-terminal graph with n vertices, the consistent tie-breaking scheme can be constructed in polynomial (in n) time [75].

Let G be a k-terminal DAG. Given a tie-breaking scheme π, the first step to construct a reachability-preserving minor is to start with an empty graph H and then for every pair $p \in P$, repeatedly add the shortest-path $\pi(p)$ to H. We can alternatively think of this as deleting vertices and edges that do not participate in any shortest path among terminal-pairs in P with respect to the scheme π. Clearly, the DAG $H = (V', E')$, $E' := \pi(P)$, is a minor of G and preserves all reachability information among pairs in P. We next review the notion of a branching event, which will be useful to bound the size of H.

Definition 8.3.4 (Branching Event)**.** *A branching event is a set of two distinct directed edges $\{e_1 = (u_1, v), e_2 = (u_2, v)\}$ that enter the same node v.*

Lemma 8.3.5. *The DAG H has at most $|P|(|P| - 1|)/2$ branching events.*

Proof. First, note that by construction of H, we can associate each edge $e \in E'$ with some pair $p \in P$ such that $e \in \pi(p)$. To prove the lemma, it suffices to show that for any two terminal-pairs $p_1, p_2 \in P$, there is at most one branching event in the graph induced by $\pi(p1) \cup \pi(p_2)$. Suppose towards contradiction that there exist two terminal pairs p_1, p_2 that have two branching events in $\pi(p1) \cup \pi(p2)$. More specifically, we assume there exist two branching events

$$b := \{e_1 = (u_1, v), e_2 = (u_2, v)\} \text{ and } b' := \{e_1 = (u_1', v'), e_2 = (u_2', v')\},$$

where e_i and e_i' lie on the dipath $\pi(p_i)$, for $i = 1, 2$.

Assume without loss of generality that the vertex v appears before v' in the dipath $\pi(p_1)$. We then claim that v must also appear before v' in the dipath $\pi(p_2)$, since otherwise we would have a directed cycle between v and v', thus contradicting the fact that H is acyclic. Since the tie-breaking scheme π is consistent (Lemma 8.3.3), it follows that the dipaths $\pi(p_1)$ and $\pi(p_2)$ must share the subpath $\pi(v, v')$. Thus, $\pi(p_1)$ and $\pi(p_2)$ use the same edge that enters the node v', i.e., $e_1' = e_2'$. However, by definition of a branching event, the edges that enter a node must be distinct, contradicting the fact that b' is a branching event. This implies that there cannot be two branching events for the terminal pairs p_1 and p_2, thus proving the lemma. □

We now have all the tools to present our algorithm for constructing reachability-preserving minors for DAGs.

Algorithm 8.1: MINORSPARSIFYDAG (G, P)

Input : k-terminal DAG G, pair-set P
Output : Reachability preserving minor H of G with respect to P

1 Set $H \leftarrow \emptyset$
2 Compute a consistent tie-breaking scheme π for shortest paths in G
3 For each $p \in P$, add the shortest path $\pi(p)$ to H
4 **while** *there is an edge* (u, v) *directed towards a non-terminal* v *with* $\deg^-_H(v) = 1$ **do**
5 Contract the edge (u, v)
6 **return** H

Lemma 8.3.6. *Given a k-terminal DAG G with a pair-set P, Algorithm 33 outputs a reachability-preserving minor H of size $O(|P|^2)$ for G with respect to P.*

Proof. We first argue that H is a reachability-preserving minor with respect to the terminals. Indeed, after Line 2 of the algorithm, graph H can viewed as deleting vertices and edges from G that do not lie on any of the shortest path among terminal pairs in P, chosen according to the scheme π. Thus, at this point H is clearly a minor of G that preserves the reachability information among the pairs in P. The edge contractions we perform in the remaining part of the algorithm guarantee that the resulting H remains a reachability-preserving minor of G with respect to P.

To bound the size of H, note that every non-terminal $v \in V' \setminus K$ has in-degree at least 2, and thus it corresponds to at least one branching event. Lemma 8.3.5 shows that the number of branching events is at most $O(|P|^2)$. Observing that edge contractions in Line 4 do not affect this number, we get that the size of H is $O(|P|^2)$. □

We next show how the construction of reachability-preserving minors can be reduced from general digraphs to DAGs, and prove the following theorem.

Theorem 8.3.7. *Given a k-terminal digraph G with a pair-set P, there exists a polynomial time algorithm that outputs a reachability-preserving minor H of size $O(|P|^2)$ with respect to P.*

Taking P to be the trivial pair-set, we get a reachability-preserving minor of size $O(k^4)$.

Proof of Theorem 8.3.7. Recall that a digraph is *strongly connected* if there is a directed path between all pair of vertices. We proceed by first finding a decomposition of the graph into strongly connected components (SCCs) [242]. We observe that each SCC that contains terminals can be contracted into a smaller component only on the terminals. Then contracting each SCC into a single vertex to obtain a DAG and invoking Algorithm 33 on the resulting DAG gives some intermediate reachability-preserving minor. Finally, we show that this minor can be expanded back to produce a reachability-preserving minor for the original digraph. These steps are formally given in the procedure Algorithm 34.

Algorithm 8.2: MinorSparsify (G, P)

Input : k-terminal digraph G, pair-set P
Output : Reachability preserving minor H of G with respect to P

Compute a strongly connected component decomposition $\mathcal{D}$ of G
Let f be some initially empty labelling that records the SCC of every vertex
for *each* SCC $C \in \mathcal{D}$ **do**
    if C *contains some terminal* $x \in K$ **then**
        For all $v \in C$, set $f(v) \leftarrow x$
    else
        Choose some arbitrary $u \in C$, and set $f(v) \leftarrow u$, for all $v \in C$.

// Preprocessing Step

Let $\mathcal{D}_K$ denote the set of SCCs containing terminals in G
for *all* SSC $C \in \mathcal{D}_K$ **do**
    while C *contains some non-terminal* v **do**
        Choose some directed edge (v, u) leaving v inside C, and contract v into u

Let $\hat{G} = (\hat{V}, \hat{E})$ and $\hat{\mathcal{D}}$ denote the resulting graph and the SCC decomposition

// Main Procedure

Contract each SSC in $\hat{\mathcal{D}}$ into a single vertex, producing the DAG $G' = (V', E')$
Let $K' \leftarrow \emptyset$ and $P' \leftarrow \emptyset$ be the terminal set and pair-set of G', respectively
For all $k \in K$, add $f(k)$ to K' and remove duplicates, if any
For all $(s, t) \in P$, add $(f(s), f(t))$ to P' if $f(s) \neq f(t)$
Set $H' =$MinorSparsifyDag(G', P')
Let H be the graph obtained by expanding back all contracted SCCs in $\hat{\mathcal{D}}_K$ in H'
return H

The main intuition behind the correctness of the above reduction lies on two important observations. First, vertices belonging to the same strongly connected components can always reach each other. Second, vertices belonging to different strongly connected components can reach each other if the corresponding vertices in the contracted graph can do so. We have the following useful observation.

Fact 8.3.8. *For any strongly connected digraph $G = (V, E)$, contracting any edge $e \in E$ results in another strongly connected digraph $G' = (V', E')$.*

Now we show that the graph H output by MinorSparsify is a reachability-preserving minor of G. It is easy to verify that the produced graph H is indeed a minor of G. To show the correctness, we will prove that H preserves the reachability information among all pairs from P in G. Before doing that, observe that the graph $\hat{G}$ obtained after the preprocessing step is a reachability preserving minor of G with respect to P. Indeed, this can be inferred by a repeated application of Fact 8.3.8 to each SSC containing terminal vertices.

Now, let $(s, t) \in P$ be any terminal-pair in G. Assume that t is reachable from s in G. We distinguish two cases:

1. If s and t belong to the same SCC in $\mathcal{D}$, they do also belong to the corresponding SCC in $\hat{\mathcal{D}}$. In Line 10, s and t are contracted into a single terminal. However, since the contracted SSC contains terminals, it is expanded back to its original form in $\hat{\mathcal{D}}$ in Line 17. Thus, it follows that t is reachable from s in the output graph H.

2. If s and t do not belong to the same SCC in $\mathcal{D}$, they must also not belong to the same SCC in $\hat{\mathcal{D}}$. Let $f(s)$ and $f(t)$ denote the terminals in the DAG G' obtained by contracting their corresponding components in $\hat{\mathcal{D}}$ (Line 10). Since t is reachable from s in $\hat{G}$, note that $f(t)$ must also be reachable from $f(s)$ in G'. By Lemma 8.3.6, it follows that $f(t)$ is reachable from $f(s)$ in the reachability-preserving minor H' of G'. Expanding back the SCCs that contain terminals in H' (Line 17), we can construct the directed path $s \rightsquigarrow f(s) \rightsquigarrow f(t) \rightsquigarrow t$ in H, which shows that t is also reachable from s in the output graph H.

When t is not reachable from s in G, we can similarly show that t is also not reachable from s in H, thus concluding the correctness proof.

We now bound the size of H. Since the DAG G' has $|P'| \leq |P|$ pairs, it follows by Lemma 8.3.6 that H' has size at most $O(|P|^2)$. After expanding back the SCCs in Line 19, we get that each SSC in H contains at most k_i terminals, where $k = \sum_i k_i$. Note that this does not contribute to the size of H. Therefore, we get that the size of the output graph H is at most $O(|P|^2)$. □

8.3.2 An Improved Bound of $O(k^3)$

Using the recent work due to Abboud and Bodwin [1], we next show how to get a polynomial improvement on the number of branching events from Lemma 8.3.5. This in turn gives a polynomial improvement on the size of reachability-preserving minor from Theorem 8.3.7.

Specifically, given a k-terminal DAG G with a pair-set P, let $H = (V, E')$ be the subgraph of G with minimum number of edges that preserves all reachability information among the pairs in P. We call such an H the *sparsest reachability preserver* of G. The following lemma is implicit in [1], and we include it here for the sake of completeness.

Lemma 8.3.9. *The DAG $H = (V, E')$ has at most $k \cdot |P|$ branching events.*

Proof. For each pair $(s, t) \in P$, we associate a directed path $s \rightsquigarrow t$ in H, and let $\tilde{\pi}(s, t)$ denote such a path. Note that since H is acyclic, every $\tilde{\pi}(s, t)$ is acyclic as well. Moreover, using the fact that H is the sparsest reachability preserver, it follows that for every edge $e \in E'$, there must be some pair $(s, t) \in P$ such that deleting e from H implies that s cannot reach t, i.e., $s \not\rightsquigarrow t$ in $H \setminus \{e\}$. This naturally leads to a relationship between edges and pairs. Specifically, we say that every edge $e \in E'$ is *owned* by one such pair $(s, t) \in P$.

Next, for each $(s,t) \in P$, we let $B^H_{(s,t)}$ denote the set of all branching events $\{e_1, e_2\}$ in H such that either e_1 or e_2 (but not both) is owned by (s,t). We claim that $\bigcup\{B^H_{(s,t)} \mid (s,t) \in P\}$ contains all branching events in H. Indeed, suppose towards contradiction that $\{e_1, e_2\}$ is a branching event in H but not in $\bigcup\{B^H_{(s,t)} \mid (s,t) \in P\}$. Then by definition of $B^H_{(s,t)}$ there must be some pair $(s,t) \in P$ such that e_1 and e_2 are both owned by (s,t). The latter implies that we can construct two directed paths from s to t, where one path uses e_1 and the other uses e_2. Delete edge e_1 w.l.o.g. Then we still have another directed path from s to t, thus contradicting the assumption that e_1 is owned by (s,t).

Now, to prove the lemma it suffices to show that $|B^H_{(s,t)}| \leq k$, for every $(s,t) \in P$. Suppose towards contradiction that there exists a pair $(s,t) \in P$ such that $|B^H_{(s,t)}| \geq k+1$. Then by the pigeonhole principle, there exist two branching events

$$\{(x_1, b_1), (x_2, b_1)\},\ \{(y_1, b_2), (y_2, b_2)\} \in B^H_{(s,t)}$$

entering the nodes b_1 and b_2, such that (s,t) owns (x_1, b_1) and (y_1, b_2), and the other edges are owned by pairs that share a common left terminal, i.e.,

$$(x_2, b_1) \text{ is owned by } (u, v_1) \text{ and } (y_2, b_2) \text{ is owned by } (u, v_2)$$

for some $u \in K$ and $(u, v_1), (u, v_2) \in P$. Note that by the definition of $B^H_{(s,t)}$, y_1 and y_2 are different vertices. We further assume w.l.o.g. that node b_1 appears before b_2 in $\tilde{\pi}(s,t)$. Now, since the pair (u, v_2) owns the edge (y_2, b_2), every path $u \rightsquigarrow v_2$ must use the edge (y_2, b_2), which further implies that every path $u \rightsquigarrow b_2$ must use the edge (y_2, b_2). We can form a path $u \rightsquigarrow b_2$ by first taking the path $\tilde{\pi}(u, v_1)[u \rightsquigarrow b_1]$[1] and then extend it by concatenating it with the path $\tilde{\pi}(s,t)[b_1 \rightsquigarrow b_2]$. This implies one of the following cases: (1) $(y_2, b_2) \in \tilde{\pi}(s,t)[b_1 \rightsquigarrow b_2]$ or (2) $(y_2, b_2) \in \tilde{\pi}(u, v_1)[u, b_1]$. We show that (2) cannot happen, thus only (1) holds. To this end, suppose towards contradiction that $(y_2, b_2) \in \tilde{\pi}(u, v_1)[u, b_1]$. Then we can find a directed path $b_2 \rightsquigarrow b_1$. But since b_1 appears before b_2, we get the cycle $b_2 \rightsquigarrow b_1 \rightsquigarrow b_2$, which contradicts the fact that H is acyclic.

Finally, case (1) implies that $(y_2, b_2) \in \tilde{\pi}(s,t)$. Therefore, the path $\tilde{\pi}(s,t)$ contains both (y_1, b_2) and (y_2, b_2). On the other hand, since $\tilde{\pi}(s,t)$ is acyclic, there cannot be two vertices entering b_2, which is a contradiction. □

[1]Let $x, y, x', y' \in V$, $\tilde{\pi}(x,y)$ be a directed path from x to y, and suppose $x', y' \in \tilde{\pi}(x,y)$ with x' appearing before y'. Then $\tilde{\pi}(x,y)[x' \rightsquigarrow y']$ denotes the directed subpath from x' to y' in $\tilde{\pi}(x,y)$.

The above lemma leads to the following algorithm.

Algorithm 8.3: MINORSPARSIFYDAG2 (G, P)

Input : k-terminal DAG G, pair-set P
Output : Reachability preserving minor H of G with respect to P

Set $H = (V, E')$ be the sparsest reachability preserver with respect to P
Remove isolated non-terminal vertices from H, if any
For each $p \in P$, add the shortest path $\pi(p)$ to H
while *there is an edge* (u, v) *directed towards a non-terminal* v *with* $\deg_H^-(v) = 1$ **do**
    Contract the edge (u, v)
return H

We remark that the above construction is built upon the sparest reachability preserver H, which we can find in exponential time (say, by a brute-force approach). By using similar arguments as in the proof of Lemma 8.3.6 and Theorem 8.3.7, we have the following guarantees.

Lemma 8.3.10. *Given a k-terminal DAG G with a pair-set P, Algorithm 35 outputs a reachability-preserving minor H of size $O(k \cdot |P|)$ for G with respect to P.*

Theorem 8.3.11. *Given a k-terminal digraph G with a pair-set P, there exists an algorithm that outputs a reachability-preserving minor H of size $O(k \cdot |P|)$ with respect to P.*

Taking P to be the trivial pair-set we get a reachability-preserving minor of size $O(k^3)$, which proves Theorem 8.1.1. We note that in contrast to Theorem 8.3.7, the above theorem guarantees only an exponential-time algorithm in the worst-case. As discussed above, this comes from the assumption that we have access to the sparsest reachability preserver. It is conceivable that a similar approach that appears in [1] could be employed to achieve a better running-time. However, the focus of our work is on optimizing the size of reachability-preserving minors.

8.4 Reachability-Preserving Minors for Planar Digraphs

In this section we show that any k-terminal planar digraph G admits a reachability-preserving minor of size $O(k^2 \log k)$ and thus prove Theorem 8.1.2. This matches the lower-bound of Theorem 8.1.3 up to an $O(\log k)$ factor. The main idea is as follows. Given a k-terminal planar digraph G with the trivial pair-set P, $|P| = k(k-1)$, our goal will be to slightly increase the number of terminals while considerably reducing the size of the pair-set P, under the condition that no reachability information is lost among the terminal-pairs in P.

Preprocessing Step. Given a k-terminal digraph G, we apply Theorem 8.1.1 to get a reachability-preserving minor G'. To simplify the notation, we will use G instead of G', i.e., throughout we assume that G has at most $O(k^3)$ vertices.

Decomposition into Path-Separable Digraphs and the Algorithm. We say that a graph $G = (V, E)$ admits an *α-separator* if there exists a set $S \subset V$ whose removal partitions G into connected components, each of size at most $\alpha \cdot |V|$, where $1/2 \leq \alpha < 1$. If the vertices of S consist of the union over r paths of G, for some $r \geq 1$, we say that G is *(α, r)-path separable*. We now review the following reduction due to Thorup [245].

Theorem 8.4.1 ([245]). *Given a digraph G, we can construct a series of digraphs $G_0, \ldots, G_b$ for some $b \leq n$ such that the number of vertices and edges over all G_i's is linear in the number of vertices and edges in G, and*

1. *Each vertex and edge of G appears in at most two G_i's.*
2. *For all $u, v \in V$, if there is a dipath R from u to v in G, there is a G_i that contains R.*
3. *Each $G_i = (V_i, E_i)$ is $(1/2, 6)$-path separable.*
4. *Each G_i is a minor of G. In particular, if G is planar, so is G_i.*

Now we review how directed reachability can be efficiently represented by separator dipaths. Let G be a k-terminal directed graph G that contains some directed path Q. Assume that the vertices of Q are ordered in increasing order in the direction of Q. For each terminal $x \in K$, let $\texttt{to}_x[Q]$ be the first vertex in Q that can be reached by x, and let $\texttt{from}_x[Q]$ be the last vertex in Q that reaches x. Let (s, t) be a terminal pair and let R be the directed path from s to t in G. We say that R intersects Q iff s can reach $\texttt{to}_s[Q]$ and t can be reached from $\texttt{from}_t[Q]$ in Q, and $\texttt{to}_s[Q]$ precedes $\texttt{from}_t[Q]$ in Q.

We now are going to combine the above tools to give our labelling algorithm aimed at reducing the size of the trivial pair-set P. By Theorem 8.4.1, we restrict our attention only to the digraphs G_i. Let $K_i := V(G_i) \cap K$ be the set of terminals restricted to the graph G_i.

Lemma 8.4.2. *Let G be a k-terminal planar digraph. Let $P' := \cup_{i=0}^{b} P'_i$ be the union over all pair-sets output by running Algorithm 36 on each digraph G_i. Then the size of $|P'|$ is at most $O(k \log k)$. Moreover, if H is a reachability-preserving minor of G with respect to P', then H is a reachability-preserving minor of G with respect to all terminal pairs.*

Proof. By preprocessing, G has at most $O(k^3)$ vertices. Throughout, it will be useful to think of the above algorithm as simultaneously running it on each digraph G_i. By Item 2 of Theorem 8.4.1, each terminal appears in at most two G_i's. Thus at each recursive level, there will be at most $O(k)$ active G_i's. Also, note that the separator properties imply that there are $O(\log k)$ recursive calls overall.

We next bound the size of the pair-set P'. Let q denote the total number of newly added terminals in Line 7 per recursive level. Since there are $O(k)$ terminals, each adding at most $O(1)$ new terminals, it follows that $q = O(k)$. First, we argue about the number of pairs added in Line 9. Since this is bounded by $O(q)$, it follows that

Algorithm 8.4: ReducePairSet (G_i, K_i)

Input : planar digraph G, terminals K_i
Output : Pair-set P_i with respect to K_i

1 **if** $|V(G_i)| \leq 1$ *or* $K_i = \emptyset$ **then**
2 **return** $\emptyset$
3 Let $P_i' \leftarrow \emptyset$ be a new pair-set
4 Compute a $1/2$-separator S of G_i consisting of 6 dipaths by Item 4 of Theorem 8.4.1
5 **for** *each dipath* $Q \in S$ **do**
  // Addition of terminal connections with Q
6 Let Q' be the set of existing terminals of Q
7 **for** *each terminal* $x \in K_i$ **do**
8 Compute $\texttt{to}_x[Q]$ and $\texttt{from}_x[Q]$
9 Declare $\texttt{to}_x[Q]$ and $\texttt{from}_x[Q]$ terminals and add them to Q'
10 Add $(x, \texttt{to}_x[Q])$ and $(\texttt{from}_x[Q], x)$ to P_i'
  // Sparsification of Q using Q'
11 Remove all vertices in $Q \setminus Q'$
12 Define directed pairs (s, t), where s and t are consecutive terminals of Q', according to the ordering of Q and add all these pairs to P_i'
13 Let $(G_i^{(1)}, K_i^{(1)})$ and $(G_i^{(2)}, K_i^{(2)})$ be the resulting graphs from $G \setminus S$, where $K_i^{(1)}$ and $K_i^{(2)}$ are disjoint subsets of the terminals K separated by S
  // Note that reachability info. about terminals in S are taken care of.
14 **return** $P_i' \cup \bigcup_{j=1}^{2}$ ReducePairSet$(G_i^{(j)}, K_i^{(j)})$

there are $O(k \log k)$ pairs overall. Second, we bound the number of pairs added when sparsifying the separator paths, i.e., pair additions in Line 11. For all the separators in the same recursive level, we can write $q := \sum_i |Q_j'|$, where Q_j' denotes the set newly added terminals for some separator dipath. By Line 11, it follows that we need only $(|Q_j'| - 1)$ pairs to represent each such dipath. Thus, per recursive call, the total number of newly added pairs is $O(q) = O(k)$. Summing these overall $O(\log k)$ levels, and combining this with the previous bound, gives the claimed bound on $|P'|$.

Finally, we argue that P' is a pair-set that can recover reachability information among terminals. Fix any terminal pair (s, t) and let R be a directed path from s to t in G. By Item 2 of Theorem 8.4.1, there is some digraph G_i that contains R. Then, R must intersect with some separator dipath Q, at some level of the recursion of the above algorithm on G_i. The above argument gives that P' contains all the necessary information to give a (possibly) another directed path from s to t in G. □

Applying Theorem 8.3.11 on the digraph G with pair-set P', as defined by the above lemma, we get Theorem 8.1.2.

8.4.1 Lower-bound for Planar DAGs

In this section we prove that there exists an infinite family of k-terminal acyclic directed grids such that any reachability-preserving minor for such graphs needs $\Omega(k^2)$ non-terminals (i.e., prove Theorem 8.1.3). We achieve this by adapting the ideas of Krauthgamer et al. [170], from their lower-bound proof on distance-preserving minors for undirected graphs.

We start by defining of our lower-bound instance. Fix k such that $r = k/4$ is an integer. Construct an initially undirected $(r+1) \times (r+1)$ grid, where all the k terminals lie on the boundary, except at the corners, and declare all non-boundary vertices non-terminals. Remove the four corner vertices, and then all boundary edges connecting the terminals. Now, make the graph directed by first directing each horizontal edge from left to right, and then directing each vertical edge from top to bottom. Let G denote the resulting k-terminal directed grid. It is easy to verify that G is acyclic.

Theorem 8.4.3. *For infinitely many $k \in \mathbb{N}$ there exists a k-terminal acyclic directed grid G such that any reachability-preserving minor of G must use $\Omega(k^2)$ non-terminals.*

Proof. Let G be the k-terminal grid defined as above. Note that there are r terminals on each side of the grid. Let H be any reachability-preserving minor of G. Recall that H contains all terminal vertices from G. Furthermore, let $x_1, x_2, \ldots, x_r$ be the terminals on the left-side of the grid, ordered from top to bottom. Similarly, define $y_1, y_2, \ldots, y_r$ to be the terminals on the right-side. Note that by construction of G, for an index pair (i, j) with $i < j$, there is no directed path from x_j to y_i. Finally, define P_H^i to be the directed path from x_i to y_i in H, for $i = 1, \ldots, r$. Throughout we will refer to such paths as *horizontal*.

Claim 8.4.4. *The horizontal directed paths $P_H^1, P_H^2, \ldots, P_H^r$ are vertex disjoint in H.*

Proof. Suppose towards contradiction that there exist some i and j with $i < j$ such that P_H^i and P_H^j intersect at some vertex z in H. This implies that there are directed paths from x_i and x_j to z, and from z to y_i and y_j. The latter implies that there is a directed path from x_j to y_i in H. However, by construction of G, we know that x_j cannot reach y_i for $i < j$, contradicting the fact that H is a reachability-preserving minor of G. □

We can apply symmetric argument to the *vertical* paths in H. More specifically, define $u_1, u_2, \ldots, u_r$ to be the terminal on the top-side of the grid, order from left to right. Similarly, define $v_1, v_2, \ldots, u_r$ to be the terminals on the bottom-side. Note that by construction of G, for an index pair (i, j) with $i < j$, there is no directed path from u_j to v_i. Finally, define Q_H^i to be the directed path from u_i to v_i in H, for $i = 1, \ldots, r$. Then we get the following symmetric claim.

Claim 8.4.5. *The vertical directed paths $Q_H^1, Q_H^2, \ldots, Q_H^r$ are vertex disjoint in H.*

We next argue that all the horizontal and the vertical paths must intersect with each other.

Claim 8.4.6. *Any pair of horizontal and vertical paths P_H^i and Q_H^j intersect in H.*

Proof. Since H is a minor of G, any dipath that connects two terminals in H can be mapped back to a dipath connecting two terminals in G. Let P_i and Q_j be the corresponding dipaths in G that are obtained by expanding back the dipaths P_H^i and Q_H^j in H. By construction of G, the horizontal and vertical dipaths between terminals are unique, implying that P_i and Q_j must intersect at some vertex of G. By performing the backtracked minor-operations on this vertex yields an intersection vertex between P_H^i and Q_H^j in H. □

The last claim we need shows that no pair of horizontal and the vertical paths intersects intersect at a terminal vertex.

Claim 8.4.7. *No pair of horizontal and vertical paths P_H^i and Q_H^j intersects at a terminal vertex in G.*

Proof. Consider the terminal pairs (x_i, y_i) and (u_j, v_j) corresponding to the paths P_H^i and Q_H^j. Note that by construction of G, the set of terminals reachable from both x_i and u_j in G is $\{y_i, y_{i+1}, \ldots, y_r\} \cup \{v_j, v_{j+1}, \ldots, v_r\}$. Since H is a reachability-preserving minor of G, x_i and u_j must also be able to reach this terminal-set in H and also P_H^i and Q_H^j cannot intersect at any terminal in $\{y_1, \ldots, y_{i-1}\} \cup \{v_1, \ldots, v_{j-1}\}$. Now, suppose towards contradiction that P_H^i and Q_H^j intersect at some terminal y_k, for $k \in \{i+1, \ldots, r\}$. This implies that in the path P_H^i, there is a directed path from y_k to y_i, for $k > i$, giving a contradiction by construction of G. Furthermore, observe that P_H^i and Q_H^j cannot intersect at y_i, as otherwise we would have a directed path from y_i to v_j, which is a contradiction by construction of G. Applying a similar argument to the case when paths intersect at some terminal v_ℓ, for $k \in \{j+1, \ldots, r\}$, gives the claim. □

We know have all the necessary tools to prove the theorem. Claim 8.4.6 shows that the paths P_H^i and Q_H^j intersect in H and let $z_H^{i,j}$ denote one of the intersection vertices. Now, we must show that all these vertices are distinct. To this end, assume that $z_H^{i_1,j_1} = z_H^{i_2,j_2}$. Since these vertices belong to both $P_H^{i_1}$ and $P_H^{i_2}$, by Claim 8.4.4 we get that $i_1 = i_2$. Similarly, by Claim 8.4.5 we get that $j_1 = j_2$. Thus, we have that all vertices $z_H^{i,j}$, for $i, j = 1, 2, \ldots, r$ are distinct. Since Claim 8.4.7 implies that none of this intersection vertices is a terminal, we conclude that H must contain at least $r^2 = (k/4)^2$ non-terminals. □

8.5 An Exact Cut Sparsifier of Size $O(k^2)$

In this section we show that given a k-terminal planar graph, where all terminals lie on the same face, one can construct a quality-1 cut sparsifier of size $O(k^2)$. Note

that it suffices to consider the case when all terminals lie on the *outer* face. We first present some basic tools.

8.5.1 Basic Tools

Wye-Delta Transformations. In this section we investigate the applicability of some graph reduction techniques that aim at reducing the number of non-terminals in a k-terminal graph. We start by reviewing the so-called *Wye-Delta* operations in graph reductions. These operations consist of five basic rules, which we describe below. (See Fig. 8.1 for illustrations.)

1. *Degree-one reduction:* Delete a degree-one non-terminal and its incident edge.
2. *Series reduction:* Delete a degree-two non-terminal y and its incident edges (x, y) and (y, z), and add a new edge (x, z) of capacity $\min\{\mathbf{c}(x, y), \mathbf{c}(y, z)\}$.
3. *Parallel reduction:* Replace all parallel edges by a single edge whose capacity is the sum over all capacities of parallel edges.
4. *Wye-Delta transformation:* Let x be a degree-three non-terminal with neighbours $\delta(x) = \{u, v, w\}$. Assume w.l.o.g.[2] that for any pair $(u, v) \in \delta(x)$, $\mathbf{c}(u, x) + \mathbf{c}(v, x) \geq \mathbf{c}(w, x)$, where $w \in \delta(v) \setminus \{u, v\}$. Then we can delete x (along with all its incident edges) and add edges (u, v), (v, w) and (w, u) with capacities $(\mathbf{c}(u, x) + \mathbf{c}(v, x) - \mathbf{c}(w, x))/2$, $(\mathbf{c}(v, x) + \mathbf{c}(w, x) - \mathbf{c}(u, x))/2$ and $(\mathbf{c}(u, x) + \mathbf{c}(w, x) - \mathbf{c}(v, x))/2$, respectively.
5. *Delta-Wye transformation:* Delete the edges of a triangle connecting x, y and z, introduce a new non-terminal vertex w and add new edges (w, x), (w, y) and (w, z) with edge capacities $\mathbf{c}(x, y)+\mathbf{c}(x, z)$, $\mathbf{c}(x, y)+\mathbf{c}(y, z)$ and $\mathbf{c}(x, z)+\mathbf{c}(y, z)$ respectively.

The following lemma (which follows from the above definitions) shows that the above rules preserve exactly all terminal minimum cuts.

Lemma 8.5.1. *Let G be a k-terminal graph and G' be a k-terminal graph obtained from G by applying one of the rules $1 - 5$. Then G' is a quality-1 cut sparsifier of G.*

For our application, it will be useful to enrich the set of rules by introducing two new operations. These operations can be realized as series of the operations 1-5. (See Fig. 8.2 and 8.3 for illustrations.)

6. *Edge deletion (with vertex x):* For a degree-three non-terminal with neighbours u, v, the edge (u, v) can be deleted, if it exists. To achieve this, we use a Delta-Wye transformation followed by a series reduction.

[2]Suppose there exist a pair $(u, v) \in \delta(x)$ with $\mathbf{c}(u, x) + \mathbf{c}(v, x) < \mathbf{c}(w, x)$, where $w \in \delta(v) \setminus \{u, v\}$. Then we can simply set $\mathbf{c}(w, x) = \mathbf{c}(u, x) + \mathbf{c}(v, x)$, since any terminal minimum cut would cut the edges (u, x) and (v, x) instead of the edge (w, x).

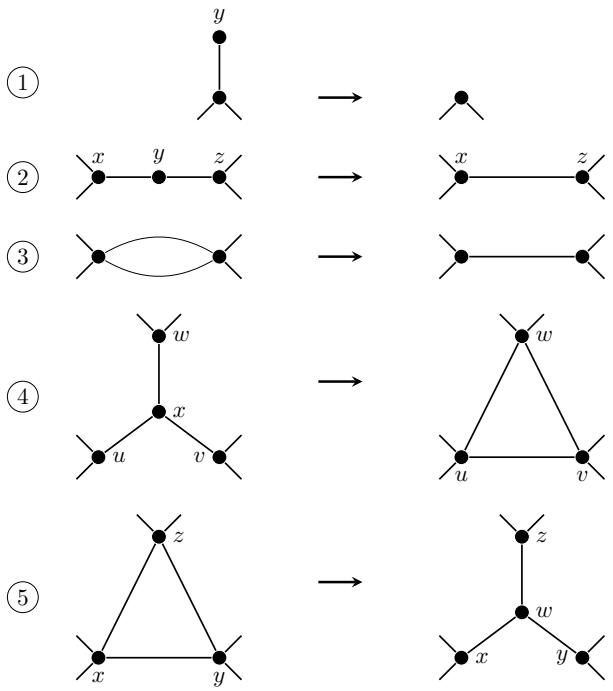

Figure 8.1: Wye-Delta operations: 1. Degree-one reduction; 2. Series reduction; 3. Parallel reduction; 4. Wye-Delta transformation; 5. Delta-Wye transformation.

7. *Edge replacement:* For a degree-four non-terminal vertex with neighbours x, u, v, w, if the edge (x, u) exists, then it can be replaced by the edge (v, w). To achieve this, we use a Delta-Wye transformation followed by a Wye-Delta transformation.

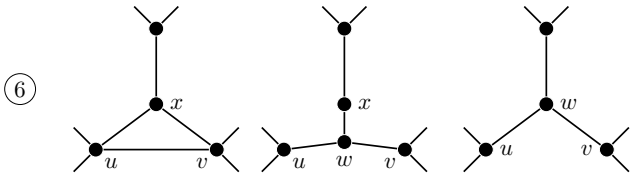

Figure 8.2: Edge deletion transformation. Edge capacities are omitted.

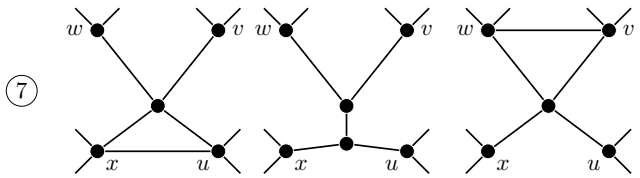

Figure 8.3: Edge replacement transformation. Edge capacities are omitted.

A k-terminal graph G is *Wye-Delta* reducible to another k-terminal graph H, if G is reduced to H by repeatedly applying one of the operations 1-7.

Lemma 8.5.2. *Let G and H be k-terminal graphs. Moreover, let G be Wye-Delta reducible to H. Then H is a quality-1 cut sparsifier of G.*

Proof. Observe that the rules 1-7 do not affect any terminal vertex and each rule preserves exactly all terminal minimum cuts by Lemma 8.5.1. An induction on the number of rules needed to reduce G to H proves the claim. □

Grid Graphs. A *grid* graph is a graph with $n \times n$ vertices $\{(u, v) : u, v = 1, \ldots, n\}$, where (u, v) and (u', v') are adjacent if $|u' - u| + |v' - v| = 1$. For $k < n$, a *half-grid* graph with k terminals is a graph $T_k^n = (V, E)$ with $K \subset V$ and $n(n+1)/2$ vertices $\{(i, j) : i \leq j \text{ and } i, j = 1, \ldots, n\}$, where (i, j) and (i', j') are connected by an edge if $|i' - i| + |j' - j| = 1$, and additional diagonal edges between (i, i) and $(i + 1, i + 1)$ for $i = 1, \ldots, n - 1$. Moreover, each terminal vertex in T_k^n must be one of its diagonal vertices, i.e., every $x \in K$ is of the form (m, m) for some $m \in \{1, \ldots, n\}$. Let $\hat{T}_k^n$ be the same graph as T_k^n but excluding the diagonal edges.

Graph Embeddings. Throughout this chapter, we will be dealing with the embedding of a planar graph into a square *grid* graph. One way of drawing graphs in the plane are *orthogonal grid-embeddings* [254]. In such a setting, the vertices correspond to distinct points and edges consist of alternating sequences of vertical and horizontal segments. Equivalently, one can view this as drawing our input graph as a subgraph of some grid. Formally, a *node-embedding* ρ of $G_1 = (V_1, E_1)$ into $G_2 = (V_2, E_2)$ is an injective mapping that maps V_1 into V_2, and E_1 into paths in G_2, i.e., (u, v) maps to a path from $\rho(u)$ to $\rho(v)$, such that every pair of paths that correspond to two different edges in G_1 is vertex-disjoint (except possibly at the endpoints). If G_2 is a planar graph, then $\rho(G_1)$ and G_1 are also planar. Thus, if G_1 and G_2 are planar we then refer to ρ as an *orthogonal embedding*. Moreover, given a planar graph G_1 drawn in the plane, the embedding ρ is called *region-preserving* if $\rho(G_1)$ and G_1 have the same planar topological embedding.

Let G_1 be a k-terminal graph. Since the embedding does not affect the vertices of G_1, the terminals of G_1 are also terminals in $\rho(G_1)$. Although the embedding

does not consider capacity of the edges in G_1, we can still guarantee that such an embedding preserves all terminal minimum cuts, for which we make use of the following operation:

1. *Edge subdivision:* Let (u, v) be an edge of capacity $\mathbf{c}(u, v)$. Delete (u, v), introduce a new vertex w and add edges (u, w) and (w, v), each of capacity $\mathbf{c}(u, v)$.

The following lemma shows that a node-embedding is a cut preserving mapping.

Lemma 8.5.3. *Let ρ be a node-embedding and let G_1 and $\rho(G_1)$ be k-terminal graphs defined as above. Then $\rho(G_1)$ preserves exactly all terminal minimum cuts of G.*

Proof. We can view each path obtained from the embedding as taking the edge corresponding to the path endpoints in G_1 and performing edge subdivisions finitely many times. We claim that such subdivisions preserve all terminal cuts.

Indeed, let us consider a single edge subdivision for (u, v) (the general claim then follows by induction on the number of edge subdivisions). Fix $S \subset K$ and consider some S-separating minimum cut $(U, V \setminus U)$ in G_1 cutting (u, v). Then, in the transformed graph $\rho(G_1)$, we can simply cut either the edge (u, w) or (w, v). Since by construction, the new edge has the same capacity as the subdivided edge, we get that $\mathrm{cap}_{\rho(G_1)}(\delta(U)) = \mathrm{cap}_{G_1}(\delta(U))$, and in particular min-cut$_{\rho(G_1)}(S, K \setminus S) \leq$ min-cut$_{G_1}(S, K \setminus S)$.

Furthermore, since G_1 is obtained by contracting two edges of the same capacity of $\rho(G_1)$, for any S-separating minimum cut $(U, V \setminus U)$ in $\rho(G_1)$, we have $\mathrm{cap}_{\rho(G_1)}(\delta(U)) \geq \mathrm{cap}_{G_1}(\delta(U))$, and in particular min-cut$_{\rho(G_1)}(S, K \setminus S) \geq$ min-cut$_{G_1}(S, K \setminus S)$. Combining the above gives the lemma. □

8.5.2 Our Construction

In this section we construct our exact cut sparsifier and prove that any planar k-terminal graph with all terminals lying on the same face admits a cut sparsifier of size $O(k^2)$ that is also planar.

Embedding into Grids

It is well-known that one can obtain an orthogonal embedding of a planar graph with maximum-degree at most three into a grid (see Valiant [254]). However, our input planar graph can have arbitrarily large maximum degree. In order to be able to make use of such an embedding, we need to first reduce our input graph to a bounded-degree graph while preserving planarity and all terminal minimum cuts. We achieve this by making use of a *vertex splitting* technique, which we describe below.

Given a k-terminal planar graph $G' = (V', E', c')$ with $K \subset V'$ lying on the outer face, vertex splitting produces a k-terminal planar graph $G = (V, E, c)$ with

$K \subset V$ such that the maximum degree of G is at most three. Specifically, for each vertex v of degree $d > 3$ with neighboring vertices $u_1, \ldots, u_d$, we delete v and introduce new vertices $v_1, \ldots, v_d$ along with edges $\{(v_i, v_{i+1}) : i = 1, \ldots, d-1\}$, each of capacity $C + 1$, where $C = \sum_{e \in E'} c'(e)$. Further, we replace the edges $\{(u_i, v) : i = 1, \ldots, d\}$ with $\{(u_i, v_i) : i = 1, \ldots, d\}$, each of corresponding capacity. If v is a terminal vertex, we set one of the v_i's to be a terminal vertex. It follows that the resulting graph G is planar and terminals can be still embedded on the outer face. Note that while the degree of every vertex v_i is at most 3, the degree of any other vertex is not affected.

Claim 8.5.4. *Let G' and G be k-terminal graphs defined as above. Then G preserves exactly all minimum terminal cuts of G', i.e., G is a quality-1 cut sparsifier of G'.*

Proof. It suffices to prove the case where G is obtained from G' by a single vertex splitting. Then the claim follows by induction on the number of vertex splittings required to transform G' to G.

Let $S \subset K$ and $(U, V \setminus U)$ be an S-separating cut in G of size min-cut$_G(S, K \setminus S)$. Suppose towards contradiction that $\delta(U)$ contains an edge of the form (v_j, v_{j+1}), for some j, which in turn gives that $\mathrm{cap}(\delta(U)) \geq C + 1$. Then we can move all the points v_i to one of the sides of the cut $(U, V \setminus S)$ and obtain a new S-separating cut in G of cost at most C, contradicting the fact that $(U, V \setminus U)$ is a minimum terminal cut. Hence, it follows that $\delta(U)$ uses either edges that are in both G and G' or edges of the form (u_i, v_i), which by construction have the same capacity as the edges (u_i, v) in G'. Thus, an S-separating minimum cut in G corresponds to an S-separating minimum cut in G' of the same cost. Since S is chosen arbitrarily, the claim follows. □

Let $G = (V, E)$ be a k-terminal graph obtained by vertex splitting of all vertices of degree larger than 3 of $G' = (V', E')$. Further, let $n' = |V'|$, $m' = |E'|$, $n = |V|$ and $m = |E|$. Then it is easy to show that $n \leq 2m'$ and $m \leq m' + n \leq 3m'$. Since G' is planar, we have that $n = O(n')$ and $m = O(n')$. Thus, by just a linear blow-up on the size of vertex and edge sets, we may assume w.l.o.g. that our input graph is a planar graph of degree at most three.

Valiant [254] and Tamassia et al. [241] showed that a k-terminal planar graph G with n vertices and degree at most three admits an orthogonal region-preserving embedding into some square grid of size $O(n) \times O(n)$. By Lemma 8.5.3, we know that the resulting graph exactly preserves all terminal minimum cuts of G. We remark that since the embedding is region-preserving, the outer face of the input graph is embedded to the outer face of the grid. Therefore, all terminals in the embedded graph lie on the outer face of the grid. Performing appropriate edge subdivisions, we can make all the terminals lie on the boundary of some possibly larger grid. Further, we can add dummy non-terminals and zero edge capacities to transform our graph into a full-grid H. We observe that the latter does not affect any terminal min-cut. The above leads to the following:

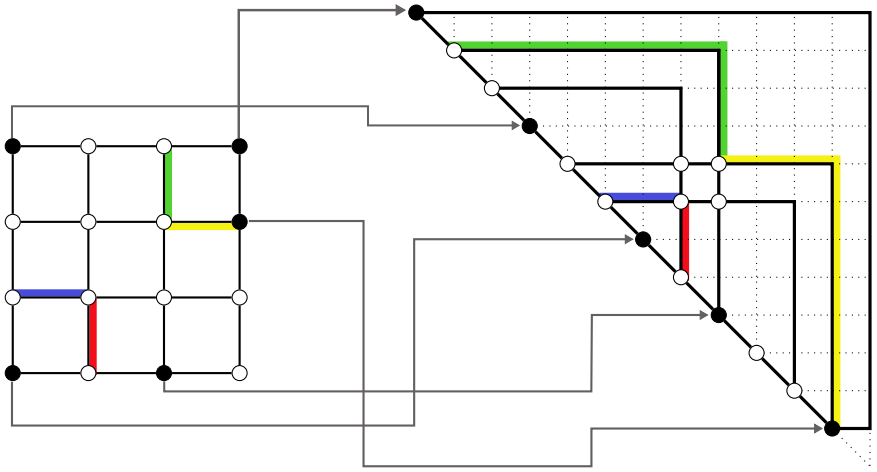

Figure 8.4: Embedding grid into half-grid. Black vertices represent terminals while white vertices represent non-terminals. The counter-clockwise ordering starts at the top right terminal. Coloured edges and paths correspond to the mapping of the respective edges: blue for edges $((i,1),(i,2))$, red for edges $((n-1,j),(n,j))$, green for edges $((1,j),(2,j))$ and yellow for edges $((i,n-1),(i,n))$, where $i,j = 2,\ldots,n-1$.

Lemma 8.5.5. *Given a k-terminal planar graph G, where all terminals lie on the outer face, there exists a k-terminal grid graph H, where all terminals lie on the boundary such that H preserves exactly all terminal minimum cuts of G. The resulting graph has $O(n^2)$ vertices and edges.*

Embedding Grids into Half-Grids

Next, we show how to embed square grids into half-grid graphs (see Section 8.2), which will facilitate the application of Wye-Delta transformations. The existence of such an embedding was claimed in the thesis of Gitler [112], but no details on its construction were given.

Let G be a k-terminal square grid on $n \times n$ vertices where terminals lie on the boundary of the grid. We obtain the following:

Lemma 8.5.6. *There exists a node embedding of the grid G into T_k^ℓ, where $\ell = 4n-3$.*

Proof. Our construction works as follows (See Fig. 8.5 for an example). We first fix an ordering on the vertices lying on the boundary of the grid in the order induced by the grid. Then we embed each vertex according to that order into the diagonal vertices of the half-grid, along with the edges that form the boundary of the grid. The sub-grid obtained by removing all boundary vertices is embedded appropriately into the upper-part of the half-grid. Finally, we show how to embed edges between

the boundary and the sub-grid vertices and argue that such an embedding is indeed vertex-disjoint for any pair of paths.

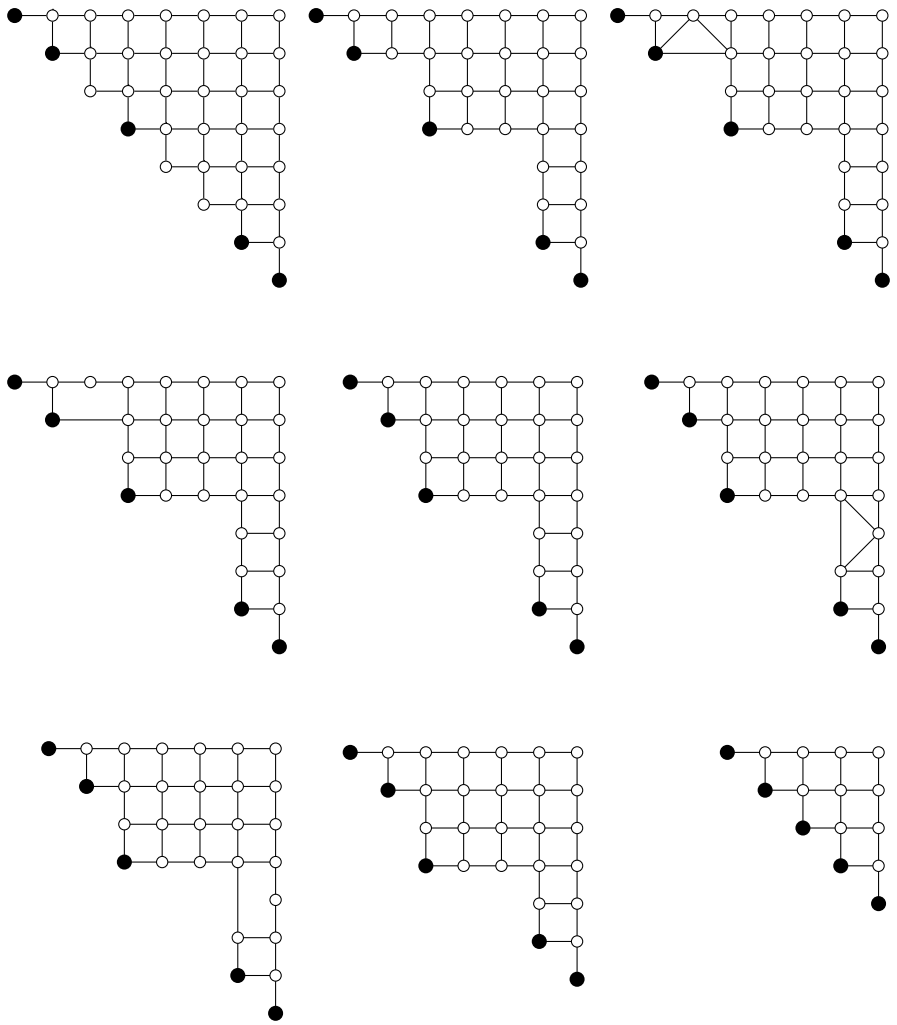

Figure 8.5: Half-Grid Reduction.

We start with the embedding of the vertices of G. Let us first consider the boundary vertices. The ordering imposed on these vertices can be viewed as starting with the upper-right vertex $(1, n)$ and visiting the rest of vertices in a counter-clockwise direction until reaching the vertex $(2, n)$. We map the vertices on the boundary as follows.

1. The vertex $(1, j)$ is mapped to the vertex $(n-j+1, n-j+1)$ for $j = 2, \ldots, n$,
2. The vertex $(i, 1)$ is mapped to the vertex $(n + i - 1, n + i - 1)$ for $i = 1, \ldots, n - 1$,

3. The vertex (n, j) is mapped to the vertex $(2n + j - 2, 2n + j - 2)$ for $j = 1, \ldots, n - 1$,
4. The vertex (i, n) is mapped to the vertex $(4n-i-2, 4n-i-2)$ for $i = 2, \ldots, n$.

Now we consider the vertices that belong to the induced sub-grid S of G of size $(n-2)^2$ when removing the boundary vertices of our input grid. We map the vertex (i, j) to the vertex $(n + i - 1, 2n + j - 2)$ for $i, j = 2, \ldots, n - 1$. In other words, for every vertex of S we make a vertical shift by $n - 1$ units and an horizontal shift by $2n - 2$ units. By construction, it is not hard to check that every vertex of G is mapped to a different vertex of T_k^ℓ and all terminal vertices lie on the diagonal of T_k^ℓ.

We continue with the embedding of the edges of G. First, every edge between two boundary vertices in G is embedded to the edge between the corresponding mapped diagonal vertices of T_k^ℓ, except the edge between $(1, n)$ and $(2, n)$. For this edge, we define an edge embedding between the corresponding vertices $(1, 1)$ and $(4n - 4, 4n - 4)$ of T_k^ℓ by using the path:

$$(1,1) \to (1,2) \to \ldots \to (1, 4n-3) \to (2, 4n-3) \\ \to \ldots \to (4n-4, 4n-3) \to (4n-4, 4n-4).$$

Next, every edge of the sub-grid S is embedded in to the edge connecting the mapped endpoints of that edge in T_k^ℓ. In other words, if (i, j) and (i', j') were connected by an edge e in S, then $(n+i-1, 2n+j-2)$ and $(n+i'-1, 2n+j'-2)$ are connected by an edge e' in T_k^ℓ and e is mapped to e'. Finally, the only edges that remain are those connecting a boundary vertex of G with a boundary vertex of S. We distinguish four cases depending on the edge position.

1. The edge $((i, 2), (i, 1))$ is mapped to the horizontal path given by:

$$(n + i - 1, 2n) \to (n + i - 1, 2n - 1) \\ \to \ldots \to (n + i - 1, n + i - 1) \text{ for } i = 2, \ldots, n - 1$$

2. The edge $((n - 1, j), (n, j))$ is mapped to the vertical path given by:

$$(2n - 2, 2n + j - 2) \to (2n - 1, 2n + j - 2) \\ \to \ldots \to (2n + j - 2, 2n + j - 2) \text{ for } j = 2, \ldots, n - 1.$$

3. The edge $((2, j), (1, j))$ is mapped to the L-shaped path:

$$(n + 1, 2n+j - 2) \to (n, 2n + j - 2) \to \ldots \to (n - j + 1, 2n + j - 2) \\ \to (n - j + 1, 2n + j - 3) \to \ldots \to (n - j + 1, n - j + 1) \\ \text{for } j = 2, \ldots, n - 1.$$

4. The edge $((i, n-1), (i, n))$ is mapped to the L-shaped path:

$$(n+i-1, 3n-3) \to (n+i-1, 3n-2) \to \ldots \to (n+i-1, 4n-i-2)$$
$$\to (n+i, 4n-i-2) \to \ldots \to (4n-i-2, 4n-i-2)$$
$$\text{for } i = 2, \ldots, n-1.$$

By construction, it follows that the paths in our edge embedding are vertex disjoint. □

Reducing Half-Grids and Bringing the Piece Together

We now review the construction of Gitler [112], which shows how to reduce half-grids to much smaller half-grids (excluding diagonal edges) whose size depends only on k. For the sake of completeness, we provide a full proof here. Recall that $\hat{T}_k^n$ is the graph T_k^n without the diagonal edges.

Lemma 8.5.7 ([112]). *For any positive k, n with $k < n$, T_k^n is Wye-Delta reducible to $\hat{T}_k^k$.*

Proof. For sake of simplicity, we assume w.l.o.g that the four vertices $(1,1)$, $(2,2)$, $(n-1, n-1)$ and (n,n) are terminals[3]. Furthermore, we say that two terminals (i,i) and (j,j) are *adjacent* iff $i < j$ and there is no terminal (ℓ, ℓ) such that $i < \ell < j$.

We next describe the reduction procedure. Also see Fig. 8.5 for an example. The reduction procedure starts by removing the diagonal edges of T_k^n, thus producing the graph $\hat{T}_k^n$. Specifically, the two edges $((1,1),(2,2))$ and $((n-1, n-1), (n,n))$ are removed using an edge deletion operation. For each remaining diagonal edge of the form $((i,i),(i+1,i+1))$, $i = 2, \ldots, n-2$ we repeatedly apply an edge replacement operation until the edge is incident to a boundary vertex $(1,j)$ or (j,n) of the grid, where an edge deletion operation with one of the neighbours of $(1,j)$ resp. (j,n) as vertex x is applied.

Now, we know that all non-terminals of the form (i,i) are degree-two vertices, thus a series reduction is applied on each of them. This produces new diagonal edges, which are effectively reduced by the above procedure. We keep removing the newly-created degree-two non-terminal vertices and the newly-created edges until no further removals are possible. At this point, the only degree-2 vertices are terminal vertices.

The resulting graph has a staircase structure, where for every pair of adjacent terminals (i,i) and (j,j), there is a non-terminal (i,j) of degree three or four, namely, the intersection vertex, and a (possibly empty) sequence of degree-three non-terminals that lie on the boundary path from (i,i) to (j,j). For $k = i+1, \ldots, j-1$, let (i,k) and (k,j) be the degree-three non-terminals lying on the row and the column subpath, respectively. Additionally, for $k = i+1, \ldots, j-1$, let

[3]If they are not terminals, we can simply define them as terminals, thus increasing the number of terminals to $k+4 = O(k)$.

$C_k^i = \{(i', k) : i' = i, \ldots, 1\}$, resp. $R_k^j = \{(k, j') : j' = j, \ldots, n\}$ be the vertices sharing the same column, resp. row with (i, k), resp. (k, j). We next show that the vertices belonging to C_k^i and R_k^j can be removed.

The removal process works as follows. For $k = i + 1, \ldots, j - 1$, we start by choosing a degree 3 vertex (i, k) and its corresponding column C_k^i. Then we apply a Wye-Delta transformation on (i, k), thus creating two new diagonal edges. Similarly as above, we remove such edges by repeatedly applying an edge replacement operation until they have been pushed to the boundary of the grid, where an edge deletion operation is applied. In the resulting graph, the vertex $(i - 1, k) \in C_k^i$ is now a degree-three non-terminal. We apply the same procedure to this vertex. Applying such a procedure to all remaining vertices of C_k^i, we eliminate a column of the grid. Symmetrically, the same process applies to the case when we want to remove the row R_k^j corresponding to the vertex (k, j).

Applying the above removal process for every adjacent terminal pair and the corresponding degree-three non-terminals, we end up with the graph $\hat{T}_k^k$, where every diagonal vertex is a terminal. By definition, it follows that $\hat{T}_k^k$ has at most $O(k^2)$ vertices. □

Combining the above reductions leads to the following theorem:

Theorem 8.5.8. *Let G be a k-terminal planar graph where all terminals lie on the outer face. Then G admits a quality-1 cut sparsifier of size $O(k^2)$, which is also a planar graph.*

Proof. Let n denote the number of vertices in G. First, we apply Lemma 8.5.5 on G to obtain a grid graph H with $O(n^2)$ vertices, which preserves exactly all terminal minimum cuts of G. We then apply Lemma 8.5.6 on H to obtain a node embedding ρ into the half-grid T_k^ℓ, where $\ell = 4n - 3$. By Lemma 8.5.3, $\rho(H)$ preserves exactly all terminal minimum cuts of H. We can further extend $\rho(H)$ to the full half-grid T_k^ℓ, if dummy non-terminals and zero edge capacities are added. Finally, we apply Lemma 8.5.7 on T_k^ℓ to obtain a Wye-Delta reduction to the reduced half-grid graph $\hat{T}_k^k$. It follows by Lemma 8.5.2 that $\hat{T}_k^k$ is a quality-1 cut sparsifier of T_k^ℓ, where the size guarantee is immediate from the definition of $\hat{T}_k^k$. □

8.6 Extensions to Planar Flow and Distance Sparsifiers

In this section we show how to extend our result for cut sparsifiers to flow and distance sparsifiers.

8.6.1 An Upper Bound for Flow Sparsifiers

We first review the notion of Flow Sparsifiers. Let $\mathbf{d}$ be a demand function over terminal pairs in G such that $\mathbf{d}(x, x') = \mathbf{d}(x', x)$ and $\mathbf{d}(x, x) = 0$ for all $x, x' \in K$. We denote by $P_{xx'}$ the set of all paths between vertices x and x', for all $x, x' \in K$.

Further, let P_e be the set of all paths using edge e, for all $e \in E$. A *concurrent* (*multi-commodity*) flow $\mathbf{f}$ of *throughput* λ is a function over terminal paths in G such that (1) $\sum_{p \in P_{xx'}} \mathbf{f}(p) \geq \lambda \mathbf{d}(x, x')$, for all distinct terminal pairs $x, x' \in K$ and (2) $\sum_{p \in P_e} \mathbf{f}(p) \leq \mathbf{c}(e)$, for all $e \in E$. We let $\lambda_G(d)$ denote the *throughput of the concurrent flow* in G that attains the largest throughput and we call a flow achieving this throughput the *maximum concurrent flow*. A graph $H = (V', E', \mathbf{c}')$, $K \subset V'$ is a *quality*-1 (*vertex*) *flow sparsifier* of G with $q \geq 1$ if for every demand function $\mathbf{d}$, $\lambda_G(\mathbf{d}) \leq \lambda_H(\mathbf{d}) \leq q \cdot \lambda_H(\mathbf{d})$.

Next we show that given a k-terminal planar graph, where all terminals lie on the outer face, one can construct a quality-1 flow sparsifier of size $O(k^2)$. Our result follows from combining the observation of Andoni et al. [24] for constructing flow-sparsifiers using flow/cut gaps and the flow/cut gap result of Okamura and Seymour [205].

Given a k-terminal graph and a demand function $\mathbf{d}$, recall that $\lambda_G(\mathbf{d})$ is the maximum fraction of $\mathbf{d}$ that can be routed in G. We define the *sparsity* of a cut $(U, V \setminus U)$ to be

$$\Phi_G(U, \mathbf{d}) := \frac{\text{cap}(\delta(U))}{\sum_{i,j:|\{i,j\} \cap U|=1} \mathbf{d}_{ij}}$$

and the *sparsest cut* as $\Phi_G(\mathbf{d}) := \min_{U \subset V} \Phi_G(U, \mathbf{d})$. Then the *flow-cut* gap is given by

$$\gamma(G) := \max\{\Phi_G(\mathbf{d})/\lambda_G(d) : \mathbf{d} \in \mathbb{R}_+^{\binom{k}{2}}\}.$$

We will make use of the following theorem:

Theorem 8.6.1 ([24]). *Given a k-terminal graph G with terminals K, let G' be a quality-β cut sparsifier for G with $\beta \geq 1$. Then for every demand function $\mathbf{d} \in \mathbb{R}_+^{\binom{k}{2}}$,*

$$\frac{1}{\gamma(G')} \leq \frac{\lambda_{G'}(\mathbf{d})}{\lambda_G(\mathbf{d})} \leq \beta \cdot \gamma(G).$$

Therefore, the graph G' with edge capacities scaled up by $\gamma(G')$ is a quality-$\beta \cdot \gamma(G) \cdot \gamma(G')$ flow sparsifier of size $|V(G')|$ for G.

This leads to the following corollary.

Corollary 8.6.2. *Let G be a k-terminal planar graph where all terminals lie on the outer face. Then G admits a quality-1 flow sparsifier of size $O(k^2)$.*

Proof. Given a k-terminal planar graph where all terminals lie on the outer face, Theorem 8.5.8 shows how to construct a cut sparsifier G' with quality $\beta = 1$ and size $O(k^2)$, which is also a planar graph with all the k terminals lying on the outer face. Okamura and Seymour [205] showed that for every k-terminal planar graph G with terminals lying on the outer face the flow-cut gap is 1. This implies that $\gamma(G) = 1$ and $\gamma(G') = 1$. Invoking Theorem 8.6.1 we get that G' is a quality-1 flow sparsifier of size $O(k^2)$ for G. □

8.6.2 An Upper Bound for Distance Sparsifiers

We first review the notion of Vertex Distance Sparsifiers. Let $G = (V, E, \mathbf{w})$ with $K \subset V$ be a k-terminal graph, where we replace the capacity function $\mathbf{c}$ with a weight or length function $\mathbf{w} : E \to \mathbb{R}_{\geq 0}$. For a terminal pair $(x, x') \in K$, let $\text{dist}_G(x, x')$ denote the shortest path with respect to the edge lengths $\mathbf{w}$ in G. A graph $H = (V', E', \mathbf{w}')$ is a *quality-q (vertex) distance sparsifier* of G with $q \geq 1$ if for any $x, x' \in K$, $\text{dist}_G(x, x') \leq \text{dist}_H(x, x') \leq q \cdot \text{dist}_G(x, x')$.

Next we argue that a symmetric approach applies to the construction of vertex sparsifiers that preserve distances. Concretely, we prove that given a k-terminal planar graph, where all terminals lie on the outer face, one can construct a quality-1 distance sparsifier of size $O(k^2)$, which is also a planar graph. It is not hard to see that almost all arguments that we used about cut sparsifiers go through, except some adaptations regarding edge lengths in the Wye-Delta rules, edge subdivision operation and vertex splitting operation.

We start adapting the Wye-Delta operations.

1. *Degree-one reduction:* Delete a degree-one non-terminal and its incident edge.
2. *Series reduction:* Delete a degree-two non-terminal y and its incident edges (x, y) and (y, z), and add a new edge (x, z) of length $\mathbf{w}(x, y) + \mathbf{w}(y, z)$.
3. *Parallel reduction:* Replace all parallel edges by a single edge whose length is the minimum over all lengths of parallel edges.
4. *Wye-Delta transformation:* Let x be a degree-three non-terminal with neighbours $\delta(x) = \{u, v, w\}$. Delete x (along with all its incident edges) and add edges $(u, v), (v, w)$ and (w, u) with lengths $\mathbf{w}(u, x) + \mathbf{w}(v, x)$, $\mathbf{w}(v, x) + \mathbf{w}(w, x)$ and $\mathbf{w}(w, x) + \mathbf{w}(u, x)$, respectively.
5. *Delta-Wye transformation:* Let x, y and z be the vertices of the triangle connecting them. Assume w.l.o.g.[4] that for any triangle edge (x, y), $\mathbf{w}(x, y) \leq \mathbf{w}(x, z) + \mathbf{w}(y, z)$, where z is the other triangle vertex. Delete the edges of the triangle, introduce a new vertex w and add new edges (w, x), (w, y) and (w, z) with edge lengths $(\mathbf{w}(x, y) + \mathbf{w}(x, z) - \mathbf{w}(y, z))/2$, $(\mathbf{w}(x, z) + \mathbf{w}(y, z) - \mathbf{w}(x, u))/2$ and $(\mathbf{w}(x, y) + \mathbf{w}(y, z) - \mathbf{w}(x, z))/2$, respectively.

The following lemma shows that the above rules preserve exactly all shortest path distances between terminal pairs.

Lemma 8.6.3. *Let G be a k-terminal graph and G' be a k-terminal graph obtained from G by applying one of the rules 1-5. Then G' is a quality-1 distance sparsifier of G.*

[4]Suppose there exists a triangle edge (x, y) with $\mathbf{w}(x, y) > \mathbf{w}(x, z) + \mathbf{w}(y, z)$, where z is the other triangle vertex. Then we can simply set $\mathbf{w}(x, y) = \mathbf{w}(x, z) + \mathbf{w}(y, z)$, since any shortest path between terminal pairs would use the edges (x, z) and (y, z) instead of the edge (x, y).

We remark that there is no need to re-define the Edge deletion and replacement operations, since they are just a combination of the above rules. An analogue of Lemma 8.5.2 can also be shown for distances. We now modify the Edge subdivision operation, which is used when dealing with graph embeddings (see Section 8.5.1).

1. *Edge subdivision*: Let (u, v) be an edge of length $\mathbf{w}(u, v)$. Delete (u, v), introduce a new vertex w and add edges (u, w) and (w, v), each of length $\mathbf{w}(u, v)/2$.

We now prove an analogue to Lemma 8.5.3.

Lemma 8.6.4. *Let ρ be a node embedding and let G_1 and $\rho(G_1)$ be k-terminal graphs as defined in Section 8.5.1. Then $\rho(G_1)$ preserves exactly all shortest path distances between terminal pairs.*

Proof. We can view each path obtained from the embedding as taking the edge corresponding to that path endpoints in G_1 and performing edge subdivisions finitely many times. We claim that such subdivisions preserve all terminal shortest paths.

Indeed, let us consider a single edge subdivison for (u, v) (the general claim then follows by induction on the number of edge subdivions). Fix $x, x' \in K$ and consider some shortest path $p(x, x')$ in G_1 that uses (u, v). We can construct in $\rho(G_1)$ a path $q(x, x')$ of the same length as follows: traverse the subpath $p(x, u)$, traverse the edges (u, w) and (w, v) and finally traverse the subpath $p(v, x')$. It follows that $\sum_{e \in p(x,x')} \mathbf{w}(e) = \sum_{e \in q(x,x')} \mathbf{w}(e)$, and thus $\text{dist}_{\rho(G_1)}(s, t) \leq \text{dist}_{G_1}(s, t)$.

On the other hand, fix $x, x' \in K$ and consider some shortest path $p'(x, x')$ in $\rho(G_1)$ that uses the two subdivided edges (u, w) and (w, v) (note that it cannot use only one of them). We can construct in G_1 a path $q'(x, x')$ of the same length as follows: traverse the subpath $p'(x, u)$, traverse the edge (u, v) and finally traverse the subpath $p'(v, x')$. It follows that $\sum_{e \in p'(x,x')} \mathbf{w}(e) = \sum_{e \in q'(x,x')} \mathbf{w}(e)$ and thus $\text{dist}_{G_1}(s, t) \leq \text{dist}_{\rho(G_1)}(s, t)$. Combining the above gives the lemma. □

We next consider vertex splitting for graphs whose maximum degree is larger than three. For each vertex v of degree $d > 3$ with $u_1, \ldots, u_d$ adjacent to v, we delete v and introduce new vertices $v_1, \ldots, v_d$ along with edges $\{(v_i, v_{i+1}) : i = 1, \ldots, d-1\}$, each of length 0. Furthermore, we replace the edges $\{(u_i, v) : i = 1, \ldots, d\}$ with $\{(u_i, v_i) : i = 1, \ldots, d\}$, each of corresponding length. If v is a terminal vertex, we make one of the v_i's be a terminal vertex. An analogue to Claim 8.5.4 gives that the resulting graph preserves all terminal shortest path distances.

We finally note that whenever we add dummy edges of capacity 0 in the cut setting, we replace them by edges of length $D + 1$ in the distance setting, where D is the sum over all edge lengths in the graph we consider. Since any shortest path in the graph does not use the added edges, the terminal shortest path remain unaffected. The above discussion leads to the following theorem.

Theorem 8.6.5. *Let G be a k-terminal planar graph where all terminals lie on the outer face. Then G admits a quality-1 distance sparsifier of size $O(k^2)$, which is also a planar graph.*

8.6.3 Incompressibility of Distances in k-Terminal Graphs

In this section we prove the following incompressibility result (i.e., Theorem 8.1.5) concerning the trade-off between quality and size of any compression function when estimating terminal distances in k-terminal graphs: for every $\varepsilon > 0$ and $t \geq 2$, there exists a (sparse) k-terminal n-vertex graph such that $k = o(n)$, and that any compression algorithm that approximates pairwise terminal distances within a factor of $t - \varepsilon$ or an additive error $2t - 3$ must use $\Omega(k^{1+1/(t-1)})$ bits. Our lower bound is inspired by the work of Matoušek [192], which has also been utilized in the context of distance oracles [251]. Our arguments rely on the recent extremal combinatorics construction (see [69]) that was used to prove lower bounds on the size of distance approximating minors.

Discussion on our result. Note that for any k-terminal graph G, if we do not have any restriction on the structure of the distance sparsifier, then G always admits a trivial quality 1 distance sparsifier H which is the complete weighted graph on k terminals with each edge weight being equal to the distance between the two endpoints in G. Furthermore, by the well-known result of Awerbuch [28], such a graph H in turn admits a multiplicative $(2t-1)$-*spanner* H' with $O(k^{1+1/t})$ edges, that is, all the distances in H are preserved up to a multiplicative factor of $2t-1$ in H', for any $t \geq 1$. This directly implies that the k-terminal graph G has a quality $2t-1$ distance sparsifier with k vertices and $O(k^{1+1/t})$ edges. On the other hand, though *unconditional* lower bounds of type similar to our result have been known for the number of edges of spanners [178, 259], we are not aware of such lower bounds for the size of *data structure* that preserves pairwise terminal distances for any k-terminal n-vertex graph when $k = o(n)$. In the extreme case when $k = n$ (i.e., all the vertices are terminals), the recent work by Abboud and Bodwin [2] shows that any data structure that preserves the distances with an additive error t needs $\Omega(n^{4/3-\varepsilon})$ bits, for any $\varepsilon > 0, t = O(n^{\delta})$ and $\delta = \delta(\varepsilon)$ (see also the follow-up work [3]).

We start by reviewing a classical notion in combinatorial design.

Definition 8.6.6 (Steiner Triple System)**.** *Given a ground set $K = [k]$, an $(3,2)$-Steiner system (abbr. $(3,2)$-SS) of K is a collection of 3-subsets of K, denoted by $\mathcal{S} = \{S_1, \ldots, S_r\}$, where $r = \binom{k}{2}/3$, such that every 2-subset of K is contained in exactly one of the 3-subsets.*

Lemma 8.6.7 ([258])**.** *For infinity many k, the set $K = [k]$ admits an $(3,2)$-SS.*

Roughly speaking, our proof proceeds by forming a k-terminal bipartite graph, where terminals lie on one side and non-terminals on the other. The set of non-terminals will correspond to some subset of a Steiner Triple System $\mathcal{S}$, which will satisfy some *certain* property. One can equivalently view such a graph as taking union over *star* graphs. Before delving into details, we need to review a couple of other useful definitions and the construction from [69].

Detour Graph and Cycle. Let k be an integer such that $K = [k]$ admits an $(3, 2)$-SS. Let $\mathcal{S}$ be such an $(3, 2)$-SS. We associate $\mathcal{S} = \{S_1, \ldots, S_r\}$ with a graph whose vertex set is $\mathcal{S}$. We refer to such graph as a *detouring graph*. By the definition of Steiner system, it follows that $|S_i \cap S_j|$ is either zero or one. Thus, two vertices S_i and S_j are adjacent in the detouring graph iff $|S_i \cap S_j| = 1$. It is also useful to label each edge (S_i, S_j) with the terminal in $S_i \cap S_j$. A *detouring cycle* is a cycle in the detouring graph such that no two neighbouring edges in the cycles have the same terminal label. Observe that the detouring graph has other cycles which are not detouring cycles.

Ideally, we would like to construct detouring graphs with long detouring cycles while keeping the size of the graph as large as possible. One trade-off is given in the following lemma.

Lemma 8.6.8 ([69]). *For any integer $t \geq 3$, given a detouring graph with vertex set $\mathcal{S}$, there exists a subset $\mathcal{S}' \subset \mathcal{S}$ of cardinality $\Omega(k^{1+1/(t-1)})$ such that the induced graph on $\mathcal{S}'$ has no detouring cycles of size t or less.*

Now we are ready to prove our incompressibility result regarding approximately preserving terminal pairwise distances.

Proof of Theorem 8.1.5: Let k be an integer such that $K = [k]$ admits an $(3, 2)$-SS $\mathcal{S}$. Fix some integer $t \geq 3$, some positive constant c and use Lemma 8.6.8 to construct a subset $\mathcal{S}'$ of $\mathcal{S}$ of size $\Omega(k^{1+1/(t-1)})$ such that the induced graph on $\mathcal{S}'$ has no detouring cycles of size t or less. We may assume w.l.o.g. that $\ell = |\mathcal{S}'| = c \cdot k^{1+1/(t-1)}$ (this can be achieved by repeatedly removing elements from $\mathcal{S}'$, as the property concerning the detouring cycles is not destroyed). Fix some ordering among 3-subsets of $\mathcal{S}'$ and among terminals in each 3-subset.

We define the k-terminal graph G as follows:

- For each $e_i \in \mathcal{S}'$ create a non-terminal vertex v_i. Let $V_{\mathcal{S}'}$ denote the set of such vertices. The vertex set of G is $K \cup V_{\mathcal{S}'}$, where $K = [k]$ denotes the set of terminals.
- For each $e_i \in \mathcal{S}'$, connect v_i to the three terminals $\{x_1^i, x_2^i, x_3^i\}$ belonging to e_i, i.e., add edges (v_i, x_j^i), $j = 1, 2, 3$.

Note that G is sparse since both the number of vertices and edges are $\Theta(\ell)$, and it also holds that $k = o(|V(G)|)$.

For any subset $R \subseteq \mathcal{S}'$, we define the subgraph $G_R = (V(G), E_R)$ of G as follows. For each $e_i \in S'$, if $e_i \in R$, perform no changes. If $e_i \notin R$, delete the edge (v_i, x_1^i). Note that there are 2^ℓ subgraphs G_R. We let $\mathcal{G}$ denote the family of all such subgraphs.

We say a terminal pair (x, x') *respects* $\mathcal{S}'$ if in the $(3, 2)$-SS $\mathcal{S}$, the unique 3-subset e that contains x and x' belongs to $\mathcal{S}'$. Given $R \subseteq \mathcal{S}'$ and some terminal pair (x, x'), we say that R *covers* (x, x') if both x and x' are connected to some non-terminal v in G_R.

Claim 8.6.9. *For all $R \subseteq \mathcal{S}'$ and terminal pairs (x, x') covered by R we have that $dist_{G_R}(x, x') = 2$.*

Proof. By the definition of Steiner system and the construction of G_R, the shortest path between x and x' is simply a 2-hop path, i.e., $\text{dist}_{G_R}(x, x') = 2$. □

Claim 8.6.10. *For all $R \subseteq \mathcal{S}'$ and any terminal pair (x, x') that respects $\mathcal{S}'$ and is not covered by R, we have that $dist_{G_R}(x, x') \geq 2t$.*

Proof. Since (x, x') respects $\mathcal{S}'$, there exists $e_i = (x_1^i, x_2^i, x_3^i) \in \mathcal{S}'$ that contains both x and x'. By construction of G_R and the fact that (x, x') is not covered by R, it follows that $e_i \in \mathcal{S}' \setminus R$, and one of x, x' corresponds to x_1^i and the other corresponds to x_2^i or x_3^i. W.l.o.g., we assume $x = x_1^i$ and $x' = x_2^i$. Note that there is no edge connecting x_1^i with the non-terminal v_i that corresponds to e_i. Now by Lemma 8.6.8, the detouring graph induced on $\mathcal{S}'$ has no detouring cycles of size t or less, which implies that any other simple path between x_1^i and x_2^i in G must pass through at least $t - 1$ other terminals. Let $w_1, \ldots, w_{t-1}$ be such terminals and let $P := x_1^i \to w_1, \ldots, w_{t-1} \to x_2^i$ denote the corresponding path, ignoring the non-terminals along the path. Between any consecutive terminal pairs in P, the shortest path is at least 2. Thus, the length of P is at least $2t$, i.e., $\text{dist}_{G_R}(x_1^i, x_2^i) \geq 2t$. □

Fix any two subsets $R_1, R_2 \subseteq \mathcal{S}'$ with $R_1 \neq R_2$. It follows that there exists a 3-subset $e_i = (x_1^i, x_2^i, x_3^i) \in \mathcal{S}'$ such that either $e \in R_1 \setminus R_2$ or $e \in R_2 \setminus R_1$. Assume w.l.o.g that $e \in R_2 \setminus R_1$. Note that (x_1^i, x_2^i) respects $\mathcal{S}'$ and it is covered in R_2 but not in R_1. By Claim 8.6.9 and 8.6.10, it holds that $\text{dist}_{G_{R_2}}(x_1^i, x_2^i) = 2$ and $\text{dist}_{G_{R_1}}(x_1^i, x_2^i) \geq 2t$. In other words, there exists a set $\mathcal{G}$ of 2^ℓ different subgraphs on the same set of nodes $V(G)$ satisfying the following property: for any $G_1, G_2 \in \mathcal{G}$, there exists a terminal pair (x, x') such that the distances between x and x' in G_1 and G_2 differ by at least a t factor as well as by at least $2t - 2$. On the other hand, for any compression function that approximates terminal path distances within a factor of $t - \varepsilon$ or an additive error $2t - 3$ and produces a bitstring with less than ℓ bits, there exist two different graphs $G_1, G_2 \in \mathcal{G}$ that map to the same bit string. Hence, any such compression function must use at least $\Omega(\ell) = \Omega(k^{1+1/(t-1)})$ bits if we want to preserve terminal distances within a $t - \varepsilon$ factor or an additive error $2t - 3$.

To complete our argument, we need to show the claim for quality $t = 2$. The only significant modification we need is the usage of an $(3, 2)$-SS in the construction of graph G (instead of using a subset of it). The remaining details are similar to the above proof and we omit them here.

8.7 Conclusion

In this chapter, we studied vertex sparsifiers for preserving reachability information, cuts, and distances. Our first contribution is studying the notion of reachability preserving minors, which are sparsifiers that preserve reachability information among

a given set of terminals and are obtained by performing minor operations on given input graphs. We show that any k-terminal planar graph admits a reachability preserving minor of size $O(k^2 \log k)$, and then prove that this result is up to a logarithmic factor in grid graphs. For general graphs we obtain an upper bound of $O(k^3)$. The algorithmic and lower bound constructions behind these results bring together techniques from reachability oracles and counting branching events in shortest path computations. Interesting open problems include closing the gap between the best-known upper and lower bounds in general graphs and improving the running time of our algorithms.

Our second contribution is studying vertex sparsifiers that preserve cuts and distances when restricted to planar graphs with terminals lying on the same faces, which are sometimes referred to as Okamura-Seymour (OS) graphs. For any k-terminal OS graph, we show that there exist quality-1 cut and distance sparsifiers that at the same time preserve planarity. The main idea behind these results is to adapt a local reduction technique, known as Why-Delta transformation, to the cut and distance measure. An important open problem is whether one can extend this technique to remove the assumption on the location of terminal vertices, or prove a non-trivial bound in the more general setting where terminals lie on a bounded number of faces, similar to Krauthgamer and Rika [172].

Bibliography

[1] Amir Abboud and Greg Bodwin. "Reachability Preservers: New Extremal Bounds and Approximation Algorithms". In: *Symposium on Discrete Algorithms (SODA)*. 2018, pp. 1865–1883.

[2] Amir Abboud and Greg Bodwin. "The 4/3 additive spanner exponent is tight". In: *Symosium on Theory of Computing (STOC)*. 2016, pp. 351–361.

[3] Amir Abboud, Greg Bodwin, and Seth Pettie. "A Hierarchy of Lower Bounds for Sublinear Additive Spanners". In: *Symposium on Discrete Algorithms (SODA)*. 2017, pp. 568–576.

[4] Amir Abboud and Søren Dahlgaard. "Popular Conjectures as a Barrier for Dynamic Planar Graph Algorithms". In: *Symposium on Foundations of Computer Science (FOCS)*. 2016, pp. 477–486.

[5] Amir Abboud, Pawel Gawrychowski, Shay Mozes, and Oren Weimann. "Near-Optimal Compression for the Planar Graph Metric". In: *Symposium on Discrete Algorithms (SODA)*. 2018, pp. 530–549.

[6] Amir Abboud and Virginia Vassilevska Williams. "Popular Conjectures Imply Strong Lower Bounds for Dynamic Problems". In: *ymposium on Foundations of Computer Science (FOCS)*. 2014, pp. 434–443.

[7] Amir Abboud and Virginia Vassilevska Williams. "Popular conjectures imply strong lower bounds for dynamic problems". In: *Symposium on Foundations of Computer Science (FOCS)*. 2014, pp. 434–443.

[8] Ittai Abraham, Yair Bartal, and Ofer Neiman. "Nearly Tight Low Stretch Spanning Trees". In: *Symposium on Foundations of Computer Science (FOCS)*. 2008, pp. 781–790.

[9] Ittai Abraham, Shiri Chechik, and Cyril Gavoille. "Fully dynamic approximate distance oracles for planar graphs via forbidden-set distance labels". In: *Symposium on Theory of Computing (STOC)*. 2012, pp. 1199–1218.

[10] Ittai Abraham, Shiri Chechik, and Sebastian Krinninger. "Fully dynamic all-pairs shortest paths with worst-case update-time revisited". In: *Symposium on Discrete Algorithms (SODA)*. 2017, pp. 440–452.

[11] Ittai Abraham, Shiri Chechik, and Kunal Talwar. "Fully Dynamic All-Pairs Shortest Paths: Breaking the $O(n)$ Barrier". In: *International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX)*. 2014, pp. 1–16.

[12] Ittai Abraham, David Durfee, Ioannis Koutis, Sebastian Krinninger, and Richard Peng. "On Fully Dynamic Graph Sparsifiers". In: *Symposium on Foundations of Computer Science (FOCS)*. 2016, pp. 335–344.

[13] Ittai Abraham and Ofer Neiman. "Using Petal-Decompositions to Build a Low Stretch Spanning Tree". In: *Symposium on Theory of Computing (STOC)*. 2012, pp. 395–406.

[14] Kook Jin Ahn and Sudipto Guha. "Graph Sparsification in the Semi-streaming Model". In: *International Colloquium on Automata Languages and Programming (ICALP)*. 2009, pp. 328–338.

[15] Alfred V. Aho, M. R. Garey, and Jeffrey D. Ullman. "The Transitive Reduction of a Directed Graph". In: *SIAM J. Comput.* 1.2 (1972), pp. 131–137.

[16] R. Aleliunas, R. J. Lipton, L. Lovasz, C. Rackoff, and R. M. Karp. "Random walks, universal traversal sequences, and the complexity of maze problems". In: *Symposium on Foundations of Computer Science (FOCS)*. 1979, pp. 218–223.

[17] Noga Alon, Richard M. Karp, David Peleg, and Douglas B. West. "A Graph-Theoretic Game and Its Application to the k-Server Problem". In: *SIAM Journal on Computing* 24.1 (1995), pp. 78–100.

[18] Stephen Alstrup, Søren Dahlgaard, Arnold Filtser, Morten Stöckel, and Christian Wulff-Nilsen. "Constructing Light Spanners Deterministically in Near-Linear Time". In: *CoRR* abs/1709.01960 (2017).

[19] Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. "Maintaining information in fully dynamic trees with top trees". In: *ACM Trans. Algorithms* 1.2 (2005), pp. 243–264.

[20] Ingo Althöfer, Gautam Das, David P. Dobkin, Deborah Joseph, and José Soares. "On Sparse Spanners of Weighted Graphs". In: *Discrete & Computational Geometry* 9 (1993), pp. 81–100.

[21] Nima Anari and Shayan Oveis Gharan. "Effective-resistance-reducing flows, spectrally thin trees, and asymmetric TSP". In: *Symposium on Foundations of Computer Science (FOCS)*. 2015, pp. 20–39.

[22] Reid Andersen, Fan R. K. Chung, and Kevin J. Lang. "Using PageRank to Locally Partition a Graph". In: *Internet Mathematics* 4.1 (2007), pp. 35–64.

[23] Reid Andersen and Uriel Feige. "Interchanging distance and capacity in probabilistic mappings". In: *CoRR* abs/0907.3631 (2009).

[24] Alexandr Andoni, Anupam Gupta, and Robert Krauthgamer. "Towards (1 + *eps*)-Approximate Flow Sparsifiers". In: *Symposium on Discrete Algorithms (SODA)*. 2014, pp. 279–293.

[25] Alexandr Andoni, Robert Krauthgamer, and Yosef Pogrow. "On Solving Linear Systems in Sublinear Time". In: *Innovations in Theoretical Computer Science Conference (ITCS)*. 2019, 3:1–3:19.

[26] Sanjeev Arora, Satish Rao, and Umesh V. Vazirani. "Expander flows, geometric embeddings and graph partitioning". In: *J. ACM* 56.2 (2009). Announced at STOC'04, 5:1–5:37.

[27] Giorgio Ausiello, Paolo Giulio Franciosa, and Giuseppe F. Italiano. "Small Stretch Spanners on Dynamic Graphs". In: *Journal of Graph Algorithms and Applications* 10.2 (2006). Announced at ESA'05, pp. 365–385.

[28] Baruch Awerbuch. "Complexity of Network Synchronization". In: *Journal of the ACM* 32.4 (1985), pp. 804–823.

[29] Greg Barnes and Uriel Feige. "Short Random Walks on Graphs". In: *SIAM Journal on Discrete Mathemathics* 9.1 (1996), pp. 19–28.

[30] Amitabh Basu and Anupam Gupta. "Steiner Point Removal in Graph Metrics". In: (2008). http://www.ams.jhu.edu/∼abasu9/papers/SPR.pdf.

[31] Surender Baswana, Manoj Gupta, and Sandeep Sen. "Fully Dynamic Maximal Matching in O(log n) Update Time". In: *SIAM J. Comput.* 44.1 (2015). Announced at FOCS'11, pp. 88–113.

[32] Surender Baswana, Ramesh Hariharan, and Sandeep Sen. "Improved decremental algorithms for maintaining transitive closure and all-pairs shortest paths". In: *J. Algorithms* 62.2 (2007), pp. 74–92.

[33] Surender Baswana, Sumeet Khurana, and Soumojit Sarkar. "Fully dynamic randomized algorithms for graph spanners". In: *ACM Transactions on Algorithms* 8.4 (2012), 35:1–35:51.

[34] Joshua Batson, Daniel A. Spielman, Nikhil Srivastava, and Shang-Hua Teng. "Spectral sparsification of graphs: theory and algorithms". In: *Communications of the ACM* 56.8 (2013), pp. 87–94.

[35] András A. Benczúr and David R. Karger. "Approximating *s-t* Minimum Cuts in $\tilde{O}(n^2)$ Time". In: *Symposium on Theory of Computing (STOC)*. 1996, pp. 47–55.

[36] András A. Benczúr and David R. Karger. "Randomized Approximation Schemes for Cuts and Flows in Capacitated Graphs". In: *SIAM Journal on Computing* 44.2 (2015), pp. 290–319.

[37] David Berman and M. S. Klamkin. "A Reverse Card Shuffle". In: *SIAM Review* 18.3 (1976), pp. 491–492.

[38] Aaron Bernstein. "Deterministic Partially Dynamic Single Source Shortest Paths in Weighted Graphs". In: *International Colloquium on Automata Languages and Programming (ICALP)*. 2017, 44:1–44:14.

[39] Aaron Bernstein. "Fully Dynamic (2 + epsilon) Approximate All-Pairs Shortest Paths with Fast Query and Close to Linear Update Time". In: *Symposium on Foundations of Computer Science (FOCS)*. 2009, pp. 693–702.

[40] Aaron Bernstein. "Maintaining Shortest Paths Under Deletions in Weighted Directed Graphs". In: *SIAM J. Comput.* 45.2 (2016), pp. 548–574.

[41] Aaron Bernstein. "Maintaining shortest paths under deletions in weighted directed graphs". In: *Symposium on Theory of Computing (STOC)*. 2013, pp. 725–734.

[42] Aaron Bernstein and Shiri Chechik. "Deterministic decremental single source shortest paths: beyond the o(mn) bound". In: *Symposium on Theory of Computing (STOC)*. 2016, pp. 389–397.

[43] Aaron Bernstein and Shiri Chechik. "Deterministic Partially Dynamic Single Source Shortest Paths for Sparse Graphs". In: *Symposium on Discrete Algorithms (SODA)*. 2017, pp. 453–469.

[44] Aaron Bernstein, Karl Däubel, Yann Disser, Max Klimm, Torsten Mütze, and Frieder Smolny. "Distance-Preserving Graph Contractions". In: *Innovations in Theoretical Computer Science Conference (ITCS)*. 2018, 51:1–51:14.

[45] Aaron Bernstein, Sebastian Forster, and Monika Henzinger. "A Deamortization Approach for Dynamic Spanner and Dynamic Maximal Matching". In: *Symposium on Discrete Algorithms (SODA)*. 2019.

[46] Aaron Bernstein and Liam Roditty. "Improved Dynamic Algorithms for Maintaining Approximate Shortest Paths Under Deletions". In: *Symposium on Discrete Algorithms (SODA)*. 2011, pp. 1355–1365.

[47] Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. "Deterministic Fully Dynamic Data Structures for Vertex Cover and Matching". In: *SIAM J. Comput.* 47.3 (2018). Announced at SODA'15, pp. 859–887.

[48] Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. "New deterministic approximation algorithms for fully dynamic matching". In: *Symposium on Theory of Computing (STOC)*. 2016, pp. 398–411.

[49] Sayan Bhattacharya, Monika Henzinger, Danupon Nanongkai, and Charalampos E. Tsourakakis. "Space- and Time-Efficient Algorithm for Maintaining Dense Subgraphs on One-Pass Dynamic Streams". In: *Symposium on Theory of Computing (STOC)*. 2015, pp. 173–182.

[50] Daniel K Blandford, Guy E Blelloch, and Ian A Kash. "Compact representations of separable graphs". In: *Symposium on Discrete Algorithms (SODA)*. 2003, pp. 679–688.

[51] 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". In: *Theory of Computing Systems* 55.3 (2014). Announced at SPAA'11, pp. 521–554.

[52] Hans L. Bodlaender, John R. Gilbert, Hjálmtýr Hafsteinsson, and Ton Kloks. "Approximating Treewidth, Pathwidth, Frontsize, and Shortest Elimination Tree". In: *J. Algorithms* 18.2 (Mar. 1995), pp. 238–255.

[53] Greg Bodwin. "Linear Size Distance Preservers". In: *Symposium on Discrete Algorithms (SODA)*. 2017, pp. 600–615.

[54] Greg Bodwin and Sebastian Krinninger. "Fully Dynamic Spanners with Worst-Case Update Time". In: *European Symposium on Algorithms (ESA)*. 2016, 17:1–17:18.

[55] Erik G. Boman and Bruce Hendrickson. "Support Theory for Preconditioning". In: *SIAM Journal on Matrix Analysis and Applications* 25.3 (2003), pp. 694–717.

[56] Karl Bringmann, Marvin Künnemann, and André Nusser. "Fréchet Distance Under Translation: Conditional Hardness and an Algorithm via Offline Dynamic Grid Reachability". In: *Symposium on Discrete Algorithms (SODA)*. 2019, pp. 2902–2921.

[57] James R Bunch and John E Hopcroft. "Triangular factorization and inversion by fast matrix multiplication". In: *Mathematics of Computation* 28.125 (1974), pp. 231–236.

[58] Hubert T.-H. Chan, Donglin Xia, Goran Konjevod, and Andréa W. Richa. "A Tight Lower Bound for the Steiner Point Removal Problem on Trees". In: *International Workshop on Approximation Algorithms for Combinatorial Optimization (APPROX)*. 2006, pp. 70–81.

[59] Hsien-Chih Chang, Pawel Gawrychowski, Shay Mozes, and Oren Weimann. "Near-Optimal Distance Emulator for Planar Graphs". In: *European Symposium on Algorithms (ESA)*. 2018, 16:1–16:17.

[60] Moses Charikar, Tom Leighton, Shi Li, and Ankur Moitra. "Vertex Sparsifiers and Abstract Rounding Algorithms". In: *Symposium on Foundations of Computer Science (FOCS)*. 2010, pp. 265–274.

[61] Shiri Chechik. "Near-Optimal Approximate Decremental All Pairs Shortest Paths". In: *Symposium on Foundations of Computer Science (FOCS)*. 2018, pp. 170–181.

[62] Shiri Chechik, Thomas Dueholm Hansen, Giuseppe F. Italiano, Jakub Lacki, and Nikos Parotsidis. "Decremental Single-Source Reachability and Strongly Connected Components in $\tilde{O}(m\sqrt{n})$ Total Update Time". In: *Symposium on Foundations of Computer Science (FOCS)*. 2016, pp. 315–324.

[63] Chandra Chekuri, Anupam Gupta, Ilan Newman, Yuri Rabinovich, and Alistair Sinclair. "Embedding k-outerplanar graphs into l1". In: *SIAM Journal on Discrete Mathematics* 20.1 (2006), pp. 119–136.

[64] Chandra Chekuri, Sanjeev Khanna, and F Bruce Shepherd. "Edge-disjoint paths in planar graphs with constant congestion". In: *SIAM Journal on Computing* 39.1 (2009), pp. 281–301.

[65] Chandra Chekuri, F Bruce Shepherd, and Christophe Weibel. "Flow-cut gaps for integer and fractional multiflows". In: *Symposium on Discrete Algorithms (SODA)*. 2010, pp. 1198–1208.

[66] Dehua Cheng, Yu Cheng, Yan Liu, Richard Peng, and Shang-Hua Teng. "Efficient sampling for Gaussian graphical models via spectral sparsification". In: *Conference on Learning Theory (COLT)*. 2015, pp. 364–390.

[67] Ho Yee Cheung, Tsz Chiu Kwok, and Lap Chi Lau. "Fast matrix rank algorithms and applications". In: *J. ACM* 60.5 (2013), 31:1–31:25.

[68] Yun Kuen Cheung. "Steiner Point Removal - Distant Terminals Don't (Really) Bother". In: *Symposium on Discrete Algorithms (SODA)*. 2018, pp. 1353–1360.

[69] Yun Kuen Cheung, Gramoz Goranci, and Monika Henzinger. "Graph Minors for Preserving Terminal Distances Approximately - Lower and Upper Bounds". In: *International Colloquium on Automata Languages and Programming (ICALP)*. 2016, 131:1–131:14.

[70] 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: *Symposium on Theory of Computing (STOC)*. 2011, pp. 273–282.

[71] Yang-Hua Chu, Sanjay G. Rao, and Hui Zhang. "A case for end system multicast". In: *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. 2000, pp. 1–12.

[72] Julia Chuzhoy. "On vertex sparsifiers with Steiner nodes". In: *Symposium on Theory of Computing (STOC)*. 2012, pp. 673–688.

[73] Julia Chuzhoy. "Routing in undirected graphs with constant congestion". In: *Symposium on Theory of Computing Conference (STOC)*. 2012, pp. 855–874.

[74] 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: *Symposium on Theory of Computing (STOC)*. 2014, pp. 343–352.

[75] Don Coppersmith and Michael Elkin. "Sparse Sourcewise and Pairwise Distance Preservers". In: *SIAM J. Discrete Math.* 20.2 (2006). Announced at SODA'05, pp. 463–501.

[76] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)* MIT Press, 2009. ISBN: 978-0-262-03384-8.

[77] Edward B Curtis, David Ingerman, and James A Morrow. "Circular planar graphs and resistor networks". In: *Linear algebra and its applications* 283.1 (1998), pp. 115–150.

[78] Søren Dahlgaard. "On the Hardness of Partially Dynamic Graph Problems and Connections to Diameter". In: *International Colloquium on Automata, Languages, and Programming (ICALP)*. 2016, 48:1–48:14.

[79] Camil Demetrescu and Giuseppe F. Italiano. "A new approach to dynamic all pairs shortest paths". In: *J. ACM* 51.6 (2004). Announced at STOC'03, pp. 968–992.

[80] Camil Demetrescu and Giuseppe F. Italiano. "Trade-offs for fully dynamic transitive closure on DAGs: breaking through the O(n^2 barrier". In: *J. ACM* 52.2 (2005). Announced at FOCS'00, pp. 147–156.

[81] Krzysztof Diks and Piotr Sankowski. "Dynamic Plane Transitive Closure". In: *European Symposium on Algorithms (ESA)*. 2007, pp. 594–604.

[82] E. A. Dinitz, A. V. Karzanov, and M. V. Lomonosov. "On the structure of a family of minimum weighted cuts in a graph". In: *Studies in Discrete Optimization* (1976), pp. 290–306.

[83] Michael Dinitz, Robert Krauthgamer, and Tal Wagner. "Towards Resistance Sparsifiers". In: *International Workshop on Approximation Algorithms for Combinatorial Optimization (APPROX)*. 2015, pp. 738–755.

[84] Yefim Dinitz and Jeffery Westbrook. "Maintaining the Classes of 4-Edge-Connectivity in a Graph On-Line". In: *Algorithmica* 20.3 (1998), pp. 242–276.

[85] Florian Dörfler and Francesco Bullo. "Kron Reduction of Graphs With Applications to Electrical Networks". In: *IEEE Trans. on Circuits and Systems* 60-I.1 (2013), pp. 150–163.

[86] Peter G Doyle and J Laurie Snell. *Random Walks and Electric Networks*. Carus Mathematical Monographs. Mathematical Association of America, 1984.

[87] David Durfee, Yu Gao, Gramoz Goranci, and Richard Peng. "Fully Dynamic Spectral Vertex Sparsifiers and Applications". In: *Symposium on Theory of Computing (STOC)*. (forthcoming). 2019.

[88] David Durfee, Rasmus Kyng, John Peebles, Anup B. Rao, and Sushant Sachdeva. "Sampling random spanning trees faster than matrix multiplication". In: *Symposium on Theory of Computing (STOC)*. 2017, pp. 730–742.

[89] David Durfee, John Peebles, Richard Peng, and Anup B Rao. "Determinant-preserving sparsification of SDDM matrices with applications to counting and sampling spanning trees". In: *Symposium on Foundations of Computer Science (FOCS)*. 2017, pp. 926–937.

[90] Michael Elkin. "Streaming and fully dynamic centralized algorithms for constructing and maintaining sparse spanners". In: *ACM Transactions on Algorithms* 7.2 (2011). Announced at ICALP'07, 20:1–20:17.

[91] Michael Elkin, Yuval Emek, Daniel A. Spielman, and Shang-Hua Teng. "Lower-Stretch Spanning Trees". In: *SIAM Journal on Computing* 38.2 (2008). Announced at STOC'05, pp. 608–628.

[92] Michael Elkin and Ofer Neiman. "Efficient Algorithms for Constructing Very Sparse Spanners and Emulators". In: *Symposium on Discrete Algorithms (SODA)*. 2017, pp. 652–669.

[93] Yuval Emek. "k-Outerplanar Graphs, Planar Duality, and Low Stretch Spanning Trees". In: *Algorithmica* 61.1 (2011), pp. 141–160.

[94] Matthias Englert, Anupam Gupta, Robert Krauthgamer, Harald Räcke, Inbal Talgam-Cohen, and Kunal Talwar. "Vertex Sparsifiers: New Results from Old Techniques". In: *SIAM J. Comput.* 43.4 (2014). Announced at APPROX'10, pp. 1239–1262.

[95] David Eppstein. "Offline Algorithms for Dynamic Minimum Spanning Tree Problems". In: *Workshop on Algorithms and Data Structures (WADS)*. 1991, pp. 392–399.

[96] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. "Sparsification - a technique for speeding up dynamic graph algorithms". In: *Journal of the ACM* 44.5 (1997). Announced at FOCS'92, pp. 669–696.

[97] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Thomas H. Spencer. "Separator Based Sparsification. I. Planary Testing and Minimum Spanning Trees". In: *J. Comput. Syst. Sci.* 52.1 (1996), pp. 3–27.

[98] Paul Erdös. "Extremal Problems in Graph Theory". In: *Theory of Graphs and its Applications (Proc. Symposium Smolenice)* (1963).

[99] Shimon Even and Yossi Shiloach. "An On-Line Edge-Deletion Problem". In: *Journal of the ACM* 28.1 (1981), pp. 1–4.

[100] Jittat Fakcharoenphol, Satish Rao, and Kunal Talwar. "A tight bound on approximating arbitrary metrics by tree metrics". In: *J. Comput. Syst. Sci.* 69.3 (2004), pp. 485–497.

[101] Thomas A Feo and J Scott Provan. "Delta-wye transformations and the efficient reduction of two-terminal planar graphs". In: *Operations Research* 41.3 (1993), pp. 572–582.

[102] Arnold Filtser. "Steiner Point Removal with Distortion $O(\log k)$". In: *Symposium on Discrete Algorithms (SODA)*. 2018, pp. 1361–1373.

[103] Tamás Fleiner and András Frank. "A quick proof for the cactus representation of mincuts". In: (2009).

[104] Randolph Ford and Delbert Fulkerson. "Maximal Flow through a Network". In: *Canadian journal of Mathematics,* 8.3 (1956), pp. 399–404.

[105] Sebastian Forster and Gramoz Goranci. "Dynamic Low-Stretch Trees via Dynamic Low-Diameter Decompositions". In: *Symposium on Theory of Computing (STOC)*. (forthcoming). 2019.

[106] Greg N. Frederickson. "Data Structures for On-Line Updating of Minimum Spanning Trees, with Applications". In: *SIAM J. Comput.* 14.4 (1985). Announced at STOC'83, pp. 781–798.

[107] Harold N. Gabow. "A Matroid Approach to Finding Edge Connectivity and Packing Arborescences". In: *Journal of Computer and System Sciences* 50.2 (1995), pp. 259–273.

[108] Harold N. Gabow. "Applications of a Poset Representation to Edge Connectivity and Graph Rigidity". In: *Symposium on Foundations of Computer Science (FOCS)*. 1991, pp. 812–821.

[109] Kshitij Gajjar and Jaikumar Radhakrishnan. "Distance-Preserving Subgraphs of Interval Graphs". In: *European Symposium on Algorithms (ESA)*. 2017, 39:1–39:13.

[110] Zvi Galil and Giuseppe F. Italiano. "Maintaining the 3-Edge-Connected Components of a Graph On-Line". In: *SIAM Journal on Computing* 22.1 (1993), pp. 11–28.

[111] Mohsen Ghaffari, Andreas Karrenbauer, Fabian Kuhn, Christoph Lenzen, and Boaz Patt-Shamir. "Near-Optimal Distributed Maximum Flow". In: *Symposium on Principles of Distributed Computing (PODC)*. 2015, pp. 81–90.

[112] Isidoro Gitler. "Delta-Wye-Delta Transformations: Algorithms and Applications". PhD thesis. Department of Combinatorics and Optimization, University of Waterloo, 1991.

[113] Gramoz Goranci, Monika Henzinger, and Pan Peng. "Dynamic Effective Resistances and Approximate Schur Complement on Separable Graphs". In: *European Symposium on Algorithms (ESA)*. 2018, 40:1–40:15.

[114] Gramoz Goranci, Monika Henzinger, and Pan Peng. "Improved Guarantees for Vertex Sparsification in Planar Graphs". In: *European Symposium on Algorithms (ESA)*. 2017, 44:1–44:14.

[115] Gramoz Goranci, Monika Henzinger, and Pan Peng. "The Power of Vertex Sparsifiers in Dynamic Graph Algorithms". In: *European Symposium on Algorithms (ESA)*. 2017, 45:1–45:14.

[116] Gramoz Goranci, Monika Henzinger, and Thatchaphol Saranurak. "Fast Incremental Algorithms via Local Sparsifiers". manuscript. 2019.

[117] Gramoz Goranci, Monika Henzinger, and Mikkel Thorup. "Incremental Exact Min-Cut in Polylogarithmic Amortized Update Time". In: *ACM Trans. Algorithms* 14.2 (2018). Announced at ESA'16, 17:1–17:21.

[118] Gramoz Goranci and Harald Räcke. "Vertex Sparsification in Trees". In: *International Workshop on Approximation and Online Algorithms (WAOA)*. 2016, pp. 103–115.

[119] Anupam Gupta. "Steiner points in tree metrics don't (really) help". In: *Symposium on Discrete Algorithms (SODA)*. 2001, pp. 220–227.

[120] Anupam Gupta and Dominick DiRenzo. "Unpublished Manuscript". In: (2016). personal communication.

[121] Anupam Gupta, Amit Kumar, and Rajeev Rastogi. "Traveling with a Pez Dispenser (or, Routing Issues in MPLS)". In: *SIAM J. Comput.* 34.2 (2004), pp. 453–474.

[122] Manoj Gupta and Shahbaz Khan. "Simple dynamic algorithms for Maximal Independent Set and other problems". In: *CoRR* abs/1804.01823 (2018).

[123] Manoj Gupta and Richard Peng. "Fully Dynamic (1+ e)-Approximate Matchings". In: *Symposium on Foundations of Computer Science (FOCS)*. 2013, pp. 548–557.

[124] Bernhard Haeupler and Jason Li. "Faster Distributed Shortest Path Approximations via Shortcuts". In: *International Symposium on Distributed Computing (DISC)*. 2018, 33:1–33:14.

[125] Torben Hagerup, Jyrki Katajainen, Naomi Nishimura, and Prabhakar Ragde. "Characterizing Multiterminal Flow Networks and Computing Flows in Networks of Small Treewidth". In: *J. Comput. Syst. Sci.* 57.3 (1998), pp. 366–375.

[126] Prahladh Harsha, Thomas P Hayes, Hariharan Narayanan, Harald Räcke, and Jaikumar Radhakrishnan. "Minimizing average latency in oblivious routing". In: *Symposium on Discrete Algorithms (SODA)*. 2008, pp. 200–207.

[127] Monika Rauch Henzinger. "A Static 2-Approximation Algorithm for Vertex Connectivity and Incremental Approximation Algorithms for Edge and Vertex Connectivity". In: *J. Algorithms* 24.1 (1997), pp. 194–220.

[128] Monika Rauch Henzinger and Valerie King. "Maintaining Minimum Spanning Forests in Dynamic Graphs". In: *SIAM J. Comput.* 31.2 (2001). Announced at ICALP'97, pp. 364–374.

[129] Monika Rauch Henzinger and Valerie King. "Randomized Fully Dynamic Graph Algorithms with Polylogarithmic Time per Operation". In: *Journal of the ACM* 46.4 (1999). Announced at STOC'95, pp. 502–516.

[130] Monika Rauch Henzinger and Mikkel Thorup. "Sampling to provide or to bound: With applications to fully dynamic graph algorithms". In: *Random Struct. Algorithms* 11.4 (1997). Announced at ICALP'96, pp. 369–379.

[131] Monika Henzinger and Valerie King. "Fully Dynamic Biconnectivity and Transitive Closure". In: *Symposium on Foundations of Computer Science (FOCS)*. 1995, pp. 664–672.

[132] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. "A Subquadratic-Time Algorithm for Dynamic Single-Source Shortest Paths". In: *Symposium on Discrete Algorithms (SODA)*. 2014, pp. 1053–1072.

[133] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. "Decremental Single-Source Shortest Paths on Undirected Graphs in Near-Linear Total Update Time". In: *Symposium on Foundations of Computer Science (FOCS)*. 2014, pp. 146–155.

[134] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. "Dynamic Approximate All-Pairs Shortest Paths: Breaking the O(mn) Barrier and Derandomization". In: *SIAM Journal on Computing* 45.3 (2016). Announced at FOCS'13, pp. 947–1006.

[135] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. "Unifying and Strengthening Hardness for Dynamic Problems via the Online Matrix-Vector Multiplication Conjecture". In: *Symposium on Theory of Computing (STOC)*. 2015, pp. 21–30.

[136] Monika Henzinger, Satish Rao, and Di Wang. "Local Flow Partitioning for Faster Edge Connectivity". In: *Symposium on Discrete Algorithms (SODA)*. 2017, pp. 1919–1938.

[137] Jacob Holm, Giuseppe F Italiano, Adam Karczmarz, Jakub Łacki, Eva Rotenberg, and Piotr Sankowski. "Contracting a planar graph efficiently". In: *European Symposium on Algorithms (ESA)*. 2017, 50:1–50:15.

[138] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. "Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity". In: *J. ACM* 48.4 (2001). Announced at STOC'98, pp. 723–760.

[139] Jacob Holm, Eva Rotenberg, and Christian Wulff-Nilsen. "Faster Fully-Dynamic Minimum Spanning Forest". In: *European Symposium on Algorithms (ESA)*. 2015, pp. 742–753.

[140] T. C. Hu. "Optimum Communication Spanning Trees". In: *SIAM Journal on Computing* 3.3 (1974), pp. 188–195.

[141] Oscar H Ibarra, Shlomo Moran, and Roger Hui. "A generalization of the fast LUP matrix decomposition algorithm and applications". In: *Journal of Algorithms* 3.1 (1982), pp. 45–56.

[142] Giuseppe F Italiano, Adam Karczmarz, Jakub Łącki, and Piotr Sankowski. "Decremental single-source reachability in planar digraphs". In: *Symposium on Theory of Computing (STOC)*. 2017, pp. 1108–1121.

[143] Giuseppe F. Italiano, Yahav Nussbaum, Piotr Sankowski, and Christian Wulff-Nilsen. "Improved algorithms for min cut and max flow in undirected planar graphs". In: *Symposium on Theory of Computing (STOC)*. 2011, pp. 313–322.

[144] Arun Jambulapati and Aaron Sidford. "Efficient $\tilde{O}(n/\varepsilon)$ Spectral Sketches for the Laplacian and its Pseudoinverse". In: *Symposium on Discrete Algorithms (SODA)*. 2018, pp. 2487–2503.

[145] Gorav Jindal, Pavel Kolev, Richard Peng, and Saurabh Sawlani. "Density Independent Algorithms for Sparsifying k-Step Random Walks". In: *International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX)*. 2017, 14:1–14:17.

[146] Lior Kamma, Robert Krauthgamer, and Huy L. Nguyen. "Cutting Corners Cheaply, or How to Remove Steiner Points". In: *SIAM J. Comput.* 44.4 (2015). Announced at SODA'14, pp. 975–995.

[147] Bruce M. Kapron, Valerie King, and Ben Mountjoy. "Dynamic graph connectivity in polylogarithmic worst case time". In: *Symposium on Discrete Algorithms (SODA)*. 2013, pp. 1131–1142.

[148] Adam Karczmarz. "Decrementai Transitive Closure and Shortest Paths for Planar Digraphs and Beyond". In: *Symposium on Discrete Algorithms (SODA)*. 2018, pp. 73–92.

[149] David Karger. "Random Sampling in Graph Optimization Problems". PhD thesis. Stanford, USA: Stanford University, 1994.

[150] David R. Karger. "Minimum cuts in near-linear time". In: *J. ACM* 47.1 (2000), pp. 46–76.

[151] David R. Karger. "Random Sampling in Cut, Flow, and Network Design Problems". In: *Mathematics of Operations Research* 24.2 (1999), pp. 383–413.

[152] David R. Karger. "Using Randomized Sparsification to Approximate Minimum Cuts". In: *Symposium on Discrete Algorithms (SODA)*. 1994, pp. 424–432.

[153] Nikolai Karpov, Marcin Pilipczuk, and Anna Zych-Pawlewicz. "An Exponential Lower Bound for Cut Sparsifiers in Planar Graphs". In: *Algorithmica* (2018).

[154] Irit Katriel, Martin Kutz, and Martin Skutella. "Reachability substitutes for planar digraphs". In: *Technical Report MPI-I-2005-1-002*. Max-Planck-Institut für Informatik, 2005.

[155] Ken-ichi Kawarabayashi, Philip N. Klein, and Christian Sommer. "Linear-Space Approximate Distance Oracles for Planar, Bounded-Genus and Minor-Free Graphs". In: *International Colloquium on Automata Languages and Programming (ICALP)*. 2011, pp. 135–146.

[156] Ken-ichi Kawarabayashi and Bruce Reed. "A separator theorem in minor-closed classes". In: *Symposium on Foundations of Computer Science (FOCS)*. 2010, pp. 153–162.

[157] Ken-ichi Kawarabayashi and Mikkel Thorup. "Deterministic Edge Connectivity in Near-Linear Time". In: *J. ACM* 66.1 (2019). Announced at STOC'15, 4:1–4:50.

[158] Jonathan A. Kelner, Yin Tat Lee, Lorenzo Orecchia, and Aaron Sidford. "An Almost-Linear-Time Algorithm for Approximate Max Flow in Undirected Graphs, and its Multicommodity Generalizations". In: *Symposium on Discrete Algorithms (SODA)*. 2014, pp. 217–226.

[159] Jonathan A. Kelner and Alex Levin. "Spectral Sparsification in the Semi-streaming Setting". In: *Theory of Computing Systems* 53.2 (2013), pp. 243–262.

[160] Jonathan A Kelner, Gary L Miller, and Richard Peng. "Faster approximate multicommodity flow using quadratically coupled flows". In: *Symposium on Theory of Computing (STOC)*. 2012, pp. 1–18.

[161] Jonathan A. Kelner, Lorenzo Orecchia, Aaron Sidford, and Zeyuan Allen Zhu. "A simple, combinatorial algorithm for solving SDD systems in nearly-linear time". In: *Symposium on Theory of Computing (STOC)*. 2013, pp. 911–920.

[162] Arindam Khan and Prasad Raghavendra. "On mimicking networks representing minimum terminal cuts". In: *Information Processing Letters* 114.7 (2014), pp. 365–371.

[163] Rohit Khandekar, Satish Rao, and Umesh V. Vazirani. "Graph partitioning using single commodity flows". In: *J. ACM* 56.4 (2009). Announced at STOC'06, 19:1–19:15.

[164] Valerie King. "Fully Dynamic Algorithms for Maintaining All-Pairs Shortest Paths and Transitive Closure in Digraphs". In: *Symposium on Foundations of Computer Science (FOCS)*. 1999, pp. 81–91.

[165] Douglas J Klein and Milan Randić. "Resistance distance". In: *Journal of mathematical chemistry* 12.1 (1993), pp. 81–95.

[166] Philip N. Klein and Sairam Subramanian. "A Fully Dynamic Approximation Scheme for Shortest Paths in Planar Graphs". In: *Algorithmica* 22.3 (1998), pp. 235–249.

[167] Ioannis Koutis, Alex Levin, and Richard Peng. "Faster Spectral Sparsification and Numerical Algorithms for SDD Matrices". In: *ACM Transactions on Algorithms* 12.2 (2016). Announced at STACS'12, 17:1–17:16.

[168] Ioannis Koutis, Gary L. Miller, and Richard Peng. "A Nearly-$m \log n$ Time Solver for SDD Linear Systems". In: *Symposium on Foundations of Computer Science (FOCS)*. 2011, pp. 590–598.

[169] Ioannis Koutis, Gary L. Miller, and Richard Peng. "Approaching Optimality for Solving SDD Linear Systems". In: *SIAM Journal on Computing* 43.1 (2014). Announced at FOCS'10, pp. 337–354.

[170] Robert Krauthgamer, Huy L. Nguyen, and Tamar Zondiner. "Preserving Terminal Distances Using Minors". In: *SIAM J. Discrete Math.* 28.1 (2014). Announced at ICALP'12, pp. 127–141.

[171] Robert Krauthgamer and Inbal Rika. "Mimicking Networks and Succinct Representations of Terminal Cuts". In: *Symposium on Discrete Algorithms (SODA)*. 2013, pp. 1789–1799.

[172] Robert Krauthgamer and Inbal Rika. "Refined Vertex Sparsifiers of Planar Graphs". In: *CoRR* abs/1702.05951 (2017).

[173] Robert Krauthgamer and Tamar Zondiner. "Preserving Terminal Distances Using Minors". In: *International Colloquium on Automata Languages and Programming (ICALP)*. 2012, pp. 594–605.

[174] Rasmus Kyng, Yin Tat Lee, Richard Peng, Sushant Sachdeva, and Daniel A. Spielman. "Sparsified Cholesky and multigrid solvers for connection laplacians". In: *Symposium on Theory of Computing (STOC)*. 2016.

[175] Rasmus Kyng, Jakub Pachocki, Richard Peng, and Sushant Sachdeva. "A Framework for Analyzing Resparsification Algorithms". In: *Symposium on Discrete Algorithms (SODA)*. 2017, pp. 2032–2043.

[176] Rasmus Kyng and Sushant Sachdeva. "Approximate Gaussian Elimination for Laplacians - Fast, Sparse, and Simple". In: *Symposium on Foundations of Computer Science (FOCS)*. 2016, pp. 573–582.

[177] Jakub Lacki and Piotr Sankowski. "Min-Cuts and Shortest Cycles in Planar Graphs in $O(n \log \log n)$ Time". In: *European Symposium on Algorithms (ESA)*. 2011, pp. 155–166.

[178] Felix Lazebnik, Vasiliy A Ustimenko, and Andrew J Woldar. "A new series of dense graphs of high girth". In: *Bulletin of the American mathematical society* 32.1 (1995), pp. 73–79.

[179] James R Lee, Manor Mendel, and Mohammad Moharrami. "A node-capacitated okamura-seymour theorem". In: *Symposium on Theory of Computing (STOC)*. 2013, pp. 495–504.

[180] Frank Thomson Leighton and Ankur Moitra. "Extensions and limits to vertex sparsification". In: *Symposium on Theory of Computing (STOC)*. 2010, pp. 47–56.

[181] Huan Li, Stacy Patterson, Yuhao Yi, and Zhongzhi Zhang. "Maximizing the Number of Spanning Trees in a Connected Graph". In: *CoRR* abs/1804.02785 (2018).

[182] Huan Li and Zhongzhi Zhang. "Kirchhoff Index As a Measure of Edge Centrality in Weighted Networks: Nearly Linear Time Algorithms". In: *Symposium on Discrete Algorithms (SODA)*. 2018, pp. 2377–2396.

[183] David Liben-Nowell and Jon M. Kleinberg. "The link prediction problem for social networks". In: *International Conference on Information and Knowledge Management (CIKM)*. 2003, pp. 556–559.

[184] Richard J. Lipton and Robert Endre Tarjan. "A Separator Theorem for Planar Graphs". In: *SIAM Journal on Applied Mathematics* 36.2 (1979), pp. 177–189.

[185] Richard Lipton, Donald Rose, and Robert Tarjan. "Generalized Nested Dissection". In: *SIAM Journal on Numerical Analysis* 16.2 (1979), pp. 346–358.

[186] Andreas Loukas and Pierre Vandergheynst. "Spectrally Approximating Large Graphs with Smaller Graphs". In: *International Conference on Machine Learning (ICML)*. Vol. 80. 2018, pp. 3243–3252.

[187] Aleksander Madry. "Computing Maximum Flow with Augmenting Electrical Flows". In: *Symposium on Foundations of Computer Science (FOCS)*. 2016, pp. 593–602.

[188] Aleksander Madry. "Fast Approximation Algorithms for Cut-Based Problems in Undirected Graphs". In: *Symposium on Foundations of Computer Science (FOCS)*. 2010, pp. 245–254.

[189] Aleksander Madry. "Navigating Central Path with Electrical Flows: From Flows to Matchings, and Back". In: *Symposium on Foundations of Computer Science (FOCS)*. 2013, pp. 253–262.

[190] Aleksander Mądry, Damian Straszak, and Jakub Tarnawski. "Fast generation of random spanning trees and the effective resistance metric". In: *Symposium on Discrete Algorithms (SODA)*. 2015, pp. 2019–2036.

[191] Konstantin Makarychev and Yury Makarychev. "Metric Extension Operators, Vertex Sparsifiers and Lipschitz Extendability". In: *Symposium on Foundations of Computer Science (FOCS)*. 2010, pp. 255–264.

[192] Jiří Matoušek. "On the distortion required for embedding finite metric spaces into normed spaces". In: *Israel Journal of Mathematics* 93.1 (1996), pp. 333–344.

[193] Karl Menger. "Zur allgemeinen kurventheorie". In: *Fundamenta Mathematicae* 1.10 (1927), pp. 96–115.

[194] Gary L. Miller and Richard Peng. "Approximate Maximum Flow on Separable Undirected Graphs". In: *Symposium on Discrete Algorithms (SODA)*. 2013, pp. 1151–1170.

[195] Gary L. Miller, Richard Peng, and Shen Chen Xu. "Parallel Graph Decompositions Using Random Shifts". In: *Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 2013, pp. 196–203.

[196] Gary L Miller, Shang-Hua Teng, William Thurston, and Stephen A Vavasis. "Separators for sphere-packings and nearest neighbor graphs". In: *Journal of the ACM (JACM)* 44.1 (1997), pp. 1–29.

[197] Ankur Moitra. "Approximation Algorithms for Multicommodity-Type Problems with Guarantees Independent of the Graph Size". In: *Symposium on Foundations of Computer Science (FOCS)*. 2009, pp. 3–12.

[198] Cameron Musco, Praneeth Netrapalli, Aaron Sidford, Shashanka Ubaru, and David P. Woodruff. "Spectrum Approximation Beyond Fast Matrix Multiplication: Algorithms and Hardness". In: 2018, 8:1–8:21.

[199] Hiroshi Nagamochi and Toshihide Ibaraki. "A Linear-Time Algorithm for Finding a Sparse k-Connected Spanning Subgraph of a k-Connected Graph". In: *Algorithmica* 7.5&6 (1992), pp. 583–596.

[200] Hiroshi Nagamochi and Toshihide Ibaraki. *Algorithmic Aspects of Graph Connectivity*. 1st ed. New York, NY, USA: Cambridge University Press, 2008.

[201] Danupon Nanongkai and Thatchaphol Saranurak. "Dynamic Cut Oracle". In: (2016). personal communication.

[202] Danupon Nanongkai and Thatchaphol Saranurak. "Dynamic spanning forest with worst-case update time: adaptive, Las Vegas, and $O(n^{1/2-\epsilon})$ time". In: *Symposium on Theory of Computing (STOC)*. 2017, pp. 1122–1129.

[203] Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. "Dynamic Minimum Spanning Forest with Subpolynomial Worst-Case Update Time". In: *Symposium on Foundations of Computer Science (FOCS)*. 2017, pp. 950–961.

[204] Ofer Neiman and Shay Solomon. "Simple Deterministic Algorithms for Fully Dynamic Maximal Matching". In: *ACM Trans. Algorithms* 12.1 (2016). Announced at STOC'13, 7:1–7:15.

[205] Haruko Okamura and Paul D. Seymour. "Multicommodity flows in planar graphs". In: *J. Comb. Theory, Ser. B* 31.1 (1981), pp. 75–81.

[206] Krzysztof Onak and Ronitt Rubinfeld. "Maintaining a large matching and a small vertex cover". In: *Symposium on Theory of Computing (STOC)*. 2010, pp. 457–464.

[207] David Peleg and Eilon Reshef. "Deterministic Polylog Approximation for Minimum Communication Spanning Trees". In: *International Colloquium on Automata, Languages and Programming*. 1998, pp. 670–681.

[208] David Peleg and Alejandro A Schäffer. "Graph spanners". In: *Journal of graph theory* 13.1 (1989), pp. 99–116.

[209] David Peleg and Eli Upfal. "The Token Distribution Problem". In: *SIAM J. Comput.* 18.2 (1989). Announced at FOCS'86, pp. 229–243.

[210] Richard Peng. "Approximate Undirected Maximum Flows in $O(m\text{polylog}(n))$ Time". In: *Symposium on Discrete Algorithms (SODA)*. 2016, pp. 1862–1867.

[211] Richard Peng, Bryce Sandlund, and Daniel Dominic Sleator. "Offline Dynamic Higher Connectivity". In: *CoRR* abs/1708.03812 (2017).

[212] Richard Peng and Daniel A. Spielman. "An efficient parallel solver for SDD linear systems". In: *Symposium on Theory of Computing (STOC)*. 2014, pp. 333–342.

[213] Johannes A. La Poutré. "Maintenance of 2- and 3-Edge-Connected Components of Graphs II". In: *SIAM Journal on Computing* 29.5 (2000), pp. 1521–1549.

[214] Harald Räcke. "Optimal hierarchical decompositions for congestion minimization in networks". In: *Symposium on Theory of Computing (STOC)*. 2008, pp. 255–264.

[215] Harald Räcke and Chintan Shah. "Improved Guarantees for Tree Cut Sparsifiers". In: *European Symposium on Algorithms (ESA)*. 2014, pp. 774–785.

[216] Harald Räcke, Chintan Shah, and Hanjo Täubig. "Computing Cut-Based Hierarchical Decompositions in Almost Linear Time". In: *Symposium on Discrete Algorithms (SODA)*. 2014, pp. 227–238.

[217] Liam Roditty. "A faster and simpler fully dynamic transitive closure". In: *ACM Trans. Algorithms* 4.1 (2008). Announced at SODA'03, 6:1–6:16.

[218] Liam Roditty, Mikkel Thorup, and Uri Zwick. "Deterministic Constructions of Approximate Distance Oracles and Spanners". In: *International Colloquium on Automata Languages and Programming (ICALP)*. 2005, pp. 261–272.

[219] Liam Roditty and Uri Zwick. "A Fully Dynamic Reachability Algorithm for Directed Graphs with an Almost Linear Update Time". In: *SIAM J. Comput.* 45.3 (2016). Announced at STOC'04, pp. 712–733.

[220] Liam Roditty and Uri Zwick. "Dynamic Approximate All-Pairs Shortest Paths in Undirected Graphs". In: *SIAM J. Comput.* 41.3 (2012), pp. 670–683.

[221] Liam Roditty and Uri Zwick. "Improved Dynamic Reachability Algorithms for Directed Graphs". In: *Symposium on Foundations of Computer Science (FOCS)*. 2002, p. 679.

[222] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. "The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing". In: *PVLDB* 11.4 (2017), pp. 420–431.

[223] Piotr Sankowski. "Dynamic Transitive Closure via Dynamic Matrix Inverse (Extended Abstract)". In: *Symposium on Foundations of Computer Science (FOCS)*. 2004, pp. 509–517.

[224] Piotr Sankowski. "Subquadratic Algorithm for Dynamic Shortest Distances". In: *International Conference on Computing and Combinatorics (COCOON)*. 2005, pp. 461–470.

[225] Thatchaphol Saranurak and Di Wang. "Expander Decomposition and Pruning: Faster, Stronger, and Simpler". In: *Symposium on Discrete Algorithms (SODA)*. 2019, pp. 2616–2635.

[226] Michael Scharf, Gordon T. Wilfong, and Lisa Zhang. "Sparsifying network topologies for application guidance". In: *IFIP/IEEE International Symposium on Integrated Network Management (IM)*. 2015, pp. 234–242.

[227] Aaron Schild. "An almost-linear time algorithm for uniform random spanning tree generation". In: *Symposium on Theory of Computing (STOC)*. 2018, pp. 214–227.

[228] Aaron Schild, Satish Rao, and Nikhil Srivastava. "Localization of Electrical Flows". In: *Symposium on Discrete Algorithms (SODA)*. 2018, pp. 1577–1584.

[229] Jonah Sherman. "Area-convexity, l_∞ regularization, and undirected multicommodity flow". In: *Symposium on Theory of Computing (STOC)*. 2017, pp. 452–460.

[230] Jonah Sherman. "Breaking the Multicommodity Flow Barrier for O(vlog n)-Approximations to Sparsest Cut". In: *Symposium on Foundations of Computer Science (FOCS)*. 2009, pp. 363–372.

[231] Jonah Sherman. "Nearly Maximum Flows in Nearly Linear Time". In: *Symposium on Foundations of Computer Science (FOCS)*. 2013, pp. 263–269.

[232] Daniel Dominic Sleator and Robert Endre Tarjan. "A Data Structure for Dynamic Trees". In: *Journal of Computer and System Sciences* 26.3 (1983), pp. 362–391.

[233] Shay Solomon. "Fully Dynamic Maximal Matching in Constant Update Time". In: *Foundations of Computer Science (FOCS)*. 2016, pp. 325–334.

[234] Daniel A. Spielman. "Algorithms, Graph Theory, and Linear Equations in Laplacian Matrices". In: *Proceedings of the International Congress of Mathematicians*. 2010.

[235] Daniel A. Spielman and Nikhil Srivastava. "Graph Sparsification by Effective Resistances". In: *SIAM J. Comput.* 40.6 (2011), pp. 1913–1926.

[236] Daniel A. Spielman and Shang-Hua Teng. "A Local Clustering Algorithm for Massive Graphs and Its Application to Nearly Linear Time Graph Partitioning". In: *SIAM J. Comput.* 42.1 (2013), pp. 1–26.

[237] Daniel A. Spielman and Shang-Hua Teng. "Nearly Linear Time Algorithms for Preconditioning and Solving Symmetric, Diagonally Dominant Linear Systems". In: *SIAM J. Matrix Analysis Applications* 35.3 (2014). Announced at STOC'04, pp. 835–885.

[238] Daniel A. Spielman and Shang-Hua Teng. "Spectral Sparsification of Graphs". In: *SIAM J. Comput.* 40.4 (2011), pp. 981–1025.

[239] Daniel A. Spielman and Jaeoh Woo. "A Note on Preconditioning by Low-Stretch Spanning Trees". In: *CoRR* abs/0903.2816 (2009). arXiv: 0903.2816.

[240] Sairam Subramanian. "A Fully Dynamic Data Structure for Reachability in Planar Digraphs". In: *European Symposium on Algorithms (ESA)*. 1993, pp. 372–383.

[241] Roberto Tamassia and Ioannis G Tollis. "Planar grid embedding in linear time". In: *IEEE Trans. Circuits Syst.* 36.9 (1989), pp. 1230–1234.

[242] Robert Endre Tarjan. "Depth-First Search and Linear Graph Algorithms". In: *SIAM J. Comput.* 1.2 (1972), pp. 146–160.

[243] Shang-Hua Teng. "The Laplacian Paradigm: Emerging Algorithms for Massive Graphs". In: *Theory and Applications of Models of Computation*. 2010, pp. 2–14.

[244] Andrew Thomason. "An extremal function for contractions of graphs". In: *Mathematical Proceedings of the Cambridge Philosophical Society* 95 (02 1984), pp. 261–265.

[245] Mikkel Thorup. "Compact oracles for reachability and approximate distances in planar digraphs". In: *J. ACM* 51.6 (2004). Announced at FOCS'01, pp. 993–1024.

[246] Mikkel Thorup. "Dynamic Graph Algorithms with Applications". In: *Scandinavian Workshop on Algorithm Theory (SWAT)*. 2000, pp. 1–9.

[247] Mikkel Thorup. "Fully-Dynamic All-Pairs Shortest Paths: Faster and Allowing Negative Cycles". In: *candinavian Workshop on Algorithm Theory (SWAT)*. 2004, pp. 384–396.

[248] Mikkel Thorup. "Fully-Dynamic Min-Cut". In: *Combinatorica* 27.1 (2007). Announced at STOC'01, pp. 91–127.

[249] Mikkel Thorup. "Near-optimal fully-dynamic graph connectivity". In: *Symposium on Theory of Computing (STOC)*. 2000, pp. 343–350.

[250] Mikkel Thorup. "Worst-case update times for fully-dynamic all-pairs shortest paths". In: *Symposium on Theorey of Computing (STOC)*. 2005, pp. 112–119.

[251] Mikkel Thorup and Uri Zwick. "Approximate distance oracles". In: *J. ACM* 52.1 (2005). Announced at STOC'01, pp. 1–24.

[252] Joel A. Tropp. "User-Friendly Tail Bounds for Sums of Random Matrices". In: *Foundations of Computational Mathemathics* 12.4 (Aug. 2012), pp. 389–434. ISSN: 1615-3375.

[253] P. M. Vaidya. "Solving Linear Equations with Symmetric Diagonally Dominant Matrices by Constructing Good Preconditioners". manuscript.

[254] Leslie G. Valiant. "Universality Considerations in VLSI Circuits". In: *IEEE Trans. Computers* 30.2 (1981), pp. 135–140.

[255] Tal Wagner, Sudipto Guha, Shiva Prasad Kasiviswanathan, and Nina Mishra. "Semi-Supervised Learning on Data Streams via Temporal Label Propagation". In: *International Conference on Machine Learning (ICML)*. 2018, pp. 5082–5091.

[256] Rephael Wenger. "Extremal graphs with no C^4's, C^6's, or C^{10}'s". In: *J. Comb. Theory, Ser. B* 52.1 (1991), pp. 113–116.

[257] Virginia Vassilevska Williams. "Multiplying matrices faster than Coppersmith-Winograd". In: *Symposium on Theory of Computing (STOC)*. 2012, pp. 887–898.

[258] Richard M. Wilson. "An Existence Theory for Pairwise Balanced Designs, III: Proof of the Existence Conjectures". In: *J. Comb. Theory, Ser. A* 18.1 (1975), pp. 71–79.

[259] David P. Woodruff. "Lower Bounds for Additive Spanners, Emulators, and More". In: *Symposium on Foundations of Computer Science (FOCS)*. 2006, pp. 389–398.

[260] Christian Wulff-Nilsen. "Fully-dynamic minimum spanning forest with improved worst-case update time". In: *Symposium on Theory of Computing (STOC)*. 2017, pp. 1130–1143.