SUBLINEAR GRAPH SPARSIFICATION WITH APPLICATIONS TO CUTS, MATCHINGS, AND FLOWS

Huan Li

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2025

Supervisor of Dissertation	Co-Supervisor of Dissertation
Sanjeev Khanna	Anindya De
Henry Salvatori Professor of Computer and Information Science	Associate Professor of Computer and Information Science

Graduate Group Chairperson

Anindya De, Associate Professor of Computer and Information Science

Dissertation Committee

Sampath Kannan, Henry Salvatori Professor of Computer and Information Science
Aaron Roth, Henry Salvatori Professor of Computer and Information Science
Erik Waingarten, Assistant Professor of Computer and Information Science
Peilin Zhong, Research Scientist of Google Research

SUBLINEAR GRAPH SPARSIFICATION WITH APPLICATIONS TO CUTS, MATCHINGS, AND FLOWS

ABSTRACT

SUBLINEAR GRAPH SPARSIFICATION WITH APPLICATIONS TO CUTS, MATCHINGS, AND FLOWS

Huan Li

Sanjeev Khanna

Anindya De

In recent years there has been a rapid increase in the volume of data across various types of computational tasks. Consequently, there has been growing interest in developing algorithms that are highly resource-efficient in terms of time, space, and communication. Although some algorithms in the existing literature are deemed efficient in the conventional sense that they require polynomial time and space in the input size, their resource demand can still be too high for computation on massive datasets. For instance, modern social networks have more than billions of edges, and thus even a linear dependence of time or space on the size of the data can be resource-prohibitive. As a result, it has been an increasingly important topic to design *sublinear* algorithms, whose resource consumption is *smaller* than the size of the data.

In this thesis, we focus on developing sublinear algorithms for datasets with graph structures, such as social networks, biological networks and Web networks, where pairwise relationship between data points is captured. We obtain close-to-optimal results for the sublinear computation of several fundamental graph problems - this involves both designing efficient sublinear algorithms and proving lower bounds showing their optimality.

The specific problems we study in this thesis include graph sparsification, hierarchical clustering, maximum matchings, maximum flows, and minimum cuts. A common and unifying theme underlying these problems is *graph sparsification*, a powerful tool that significantly reduces the size of the graph while preserving some fundamental properties of the graph. This tool can easily be seen to imply substantially more efficient algorithms, as it allows one to compress any given graph and then run any efficient algorithm on the resulting sparse representation of significantly smaller size while only incurring a negligible loss in the solution quality. The problems we study either are directly

related to graph sparsification with outputting a graph sparsifier (i.e. a sparse representation of the graph) as a goal, or require crucial use of graph sparsification as a building block in designing efficient sublinear algorithms.

Our work demonstrates the power of graph sparsification in algorithm design beyond traditional computational models, and we hope it spurs future research on graph sparsification in additional applications.

TABLE OF CONTENTS

ABSTRACT iii

LIST OF TABLES viii

LIST OF ILLUSTRATIONS ix

CHAPTER 1 : INTRODUCTION 1

1.1 Computational Models 2

1.2 Our Results 5

1.3 Roadmap 20

CHAPTER 2 : BACKGROUND 21

2.1 Notations 21

2.2 Matchings and Graph Combinatorics 21

2.3 Linear Algebra and High Dimensional Gaussian Distributions 23

2.4 Graph Sparsification 26

2.5 Capacitated Graphs and Maximum Flows 29

2.6 PRAM Primitives 32

2.7 Similarity Graphs and Hierarchical Clustering 32

CHAPTER 3 : AN IMPROVED MAXIMUM MATCHING ALGORITHM IN THE STREAMING MODEL 35

3.1 Overview of the Algorithm 35

3.2 A Matching Cover via Regularity Lemma 38

3.3 A Single-Pass Streaming Algorithm 47

CHAPTER 4 : FIRST ALGORITHMS AND LOWER BOUNDS FOR WEIGHTED GRAPH SPARSIFICATION IN THE LINEAR SKETCHING AND TURNSTILE STREAMING MODELS 55

4.1 Overview of weighted cut and spectral sparsification algorithms 55
4.2 Overview of lower bound for weighted spectral sparsification 60
4.3 A linear sketching algorithm for weighted cut sparsification 75
4.4 A linear sketching algorithm for weighted spectral sparsification 82
4.5 Preliminaries on matrix-weighted graphs 110
4.6 Almost regular graphs have only few small eigenvalues 114
4.7 Almost regular graph decomposition 119
4.8 Almost regular expander decomposition 130
4.9 Expanders are preserved under vertex sampling 149
4.10 A lower bound for weighted spectral sparsification 174

CHAPTER 5 : A PARALLEL APPROXIMATE MAXIMUM FLOW ALGORITHM IN NEAR-LINEAR WORK AND POLY-LOGARITHMIC DEPTH 180
5.1 Overview of the Algorithm 180
5.2 Recursive Computation of a Low-Quality Congestion Approximator 190
5.3 New Framework for Congestion Approximator Computation 201
5.4 Extraction of Congestion Approximators 217
5.5 Parallel Flow Decomposition by Shortcutting 226
5.6 Applications 237

CHAPTER 6 : OPTIMAL ALGORITHMS AND LOWER BOUNDS FOR SUBLINEAR TIME HIERARCHICAL CLUSTERING 245
6.1 Overview of Sublinear Time Algorithms 245
6.2 Overview of Sublinear Time Lower Bounds 249
6.3 Hierarchical Clustering using (ε, δ)-Cut Sparsification 251
6.4 Sublinear Time Algorithms 258
6.5 Sublinear Time Lower Bounds 271
6.6 Experimental Results 283

APPENDIX A : MISSING PROOFS IN CHAPTER 3 286

APPENDIX B : MISSING PROOFS IN CHAPTER 4 287

APPENDIX C : MISSING PROOFS IN CHAPTER 5 302

BIBLIOGRAPHY 333

LIST OF TABLES

TABLE 1.1 **Summary of Results.** Each row gives an upper and lower bound on the number of queries required for $\widetilde{O}(1)$ approximation in the query model. 19

TABLE 6.1 Performance of average linkage on Boston dataset (506 vertices, 95566 edges), with absolute HC cost 186085.271. 284

TABLE 6.2 Performance of complete linkage on Boston dataset (506 vertices, 95566 edges), with absolute HC cost 226485.146. 284

TABLE 6.3 Performance of average linkage on Newsgroup dataset (3516 vertices, 6178853 edges), with absolute HC cost 3960145237.881. 284

TABLE 6.4 Performance of complete linkage on Newsgroup dataset (3516 vertices, 6178853 edges), with absolute HC cost 3960674776.482. 285

LIST OF ILLUSTRATIONS

FIGURE 4.1 A block cycle graph on n vertices. Each S_i represents a block of $n^{1/5}$ vertices connected by a clique and each zigzag represents the edges of a complete bipartite graph between adjacent blocks. The red edge represents the crossing edge. All edges along the cycle have weights $n^{2/5}$, and the crossing edge has weight 1. 58
FIGURE 4.2 Model oblivious algorithm for weighted cut sparsification. 77
FIGURE 4.3 Weighted edge sampler. 80
FIGURE 4.4 Linear sketch for recovering heavy edges in G. 88
FIGURE 4.5 Linear sketch for weighted spectral sparsification. 93
FIGURE 4.6 Linear sketch for sparsifying G^{sq}. 100
FIGURE 4.7 Linear sketch for recovering heavy edges in G^{sq}. 108

FIGURE 6.1 Generation of $G \sim \mathcal{D}$. 276
FIGURE 6.2 Generation of $G' \sim \mathcal{D}_{\text{rz}}$. 278

CHAPTER 1

INTRODUCTION

Nowadays, the volume of data that are being generated and stored across the globe is escalating at a remarkable pace. Though also increasing, the computation and storing powers of modern computers still fall short of the enormous data size - either the data is too large to fit into the memory of a single computer, or the computation is too costly to be done by a single computer in a timely manner. As such, many previously deemed efficient algorithms, such as those that run in polynomial time and space in the input size, are no longer sufficient for today's purposes, and significantly more resource-efficient algorithms are sought. In particular, it has been a popular trend among algorithm researchers to design *sublinear* algorithms, whose resource consumption is *smaller* than the size of the data.

Based on the computational resource of interest, algorithm researchers have been studying the design of sublinear algorithms in various models of computation. The representative models of sublinear computation include:

1. **Streaming (sublinear space) model** ([FKM$^+$05]), where the data arrives as a stream in an online fashion, and we want to finish computation using a small amount of local memory, ideally substantially smaller than needed for storing the entire data;
2. **Query (sublinear time) model** ([Gol17]), where we are given query/sampling access to data and seek to finish computation using a running time substantially faster than reading the entire data;
3. **Massively Parallel Computation (sublinear communication) model** ([KSV10]), where the data is stored across multiple machines in a distributed manner, and we seek to finish the computation with small communication overhead between the machines, ideally substantially smaller than needed for communicating the entire data.

Within these computational models, we focus on designing sublinear algorithms for processing large-scale *graph-structured* datasets, which naturally capture pairwise relationships among a set of

objects, and include as examples the Web network, social networks, and biological networks. In this thesis, we conduct a rigorous theoretical study on what can be achieved for some most fundamental graph problems under sublinear resource constraint - this includes both (i) designing, analyzing, and implementing new sublinear algorithms, as well as (ii) proving lower bounds showing the limits of computation. In particular, our results in this thesis approach *optimality* of solving some most fundamental graph computation problems with sublinear resource, by providing algorithms and lower bounds that nearly match each other.

The specific fundamental graph problems we study in this thesis are graph sparsification, graph clustering, matchings in graphs, and graph cuts and flows. While being distinct topics on their own, these problems are also closely connected to each other. For instance, graph sparsification can be used as an effective preprocessing step for designing graph optimization algorithms, where it is able to compress the input graph to a much sparser representation with only negligible loss in the solution quality; algorithms for finding small/sparse graph cuts can in turn be used as intermediate steps of graph clustering algorithms to partition an input graph into components that are well-connected inside but sparsely connected to each other.

By presenting our technical results, our work demonstrates the power of graph sparsification as an algorithmic tool beyond traditional computational models, where the running time was the only concern. We hope our techniques spur future research on graph sparsification in additional applications.

In the rest of this chapter, we will first define the computational models that we will study. Then, we will introduce the fundamental graph problems we study and present our main technical results for each of them. Finally, we wrap up this chapter by presenting the roadmap for the rest of the thesis.

1.1. Computational Models

In this section, we introduce, in the context of graphs, the modern computational models in which we design our sublinear algorithms. In addition to the computational models we discussed above, we will also consider a few other closely related computational models. Throughout, we will use

n, m to denote the number of vertices and number of edges in the input graph respectively.

Streaming Model. In the streaming model [FKM+05], the input graph is specified by a stream of edges of arbitrary order (and their weights in the case of a weighted graph) in an online manner. The algorithm does not have random access to the input but instead has to process the input on the fly. At the end of the stream, the algorithm is required to output an answer to the problem we want to solve on the input graph. The goal is to solve the problem using a local memory (space) that is significantly smaller than the size of the input, i.e. $\ll n^2$.

Turnstile Streaming and Linear Sketching Models. In the turnstile streaming model [Mut05], a much more general model than the ordinary streaming model, the input weighted graph is specified by a stream of arbitrary edge weight updates, including both edge weight increments and edge weight decrements. An edge is considered deleted if its weight drops to 0. The weight change of each update will be specified along with the edge identity in the stream. However, we do not have random access to the input stream, and thus cannot look up the weight change history during the stream, unless we explicitly store the changes in our local memory. The goal is to output an answer to the problem we want to solve on the input weighted graph at the end of the stream, using space sublinear in the input size, ideally $\ll n^2$.

A linear sketch [Woo14] is specified by a (randomized) sketching matrix Φ of dimension $N \times \binom{n}{2}$ and a recovery algorithm $\mathcal{A}$. We associate the input graph G with a $\binom{n}{2}$-dimensional vector w_G with each entry being the weight of the corresponding edge slot (0 if the edge is not present). The linear sketch is then obtained by the dot product $\Phi \cdot w_G$. The final output is then determined by $\mathcal{A}(\Phi \cdot w_G)$, i.e. invoking $\mathcal{A}$ on the linear sketch $\Phi \cdot w_G$. We also call each row of Φ a *linear measurement*, and N the number of linear measurements we take.

It is known [Ind06, AGM12b, KLM+17] that if a linear sketching algorithm uses sketching matrix of dimension $N \times \binom{n}{2}$ of polynomial precision, and the recovery algorithm can be implemented in $N \operatorname{polylog}(n)$ bits of space, then it implies a turnstile streaming algorithm using space $N \operatorname{polylog}(n)$.

PRAM Model. The parallel random-access machine (PRAM) [FW78] is often considered as a standard parallel computation model, owing to its simplicity and its well-understood connections

to other models of parallel computation. A generalization of the usual (sequential) RAM model, the PRAM is a synchronous, shared-memory, multi-processor model. Within PRAM, there are several variants depending on how concurrent operations in the shared memory are handled: (in decreasing order of restrictiveness) exclusive-read-exclusive-write, concurrent-read-exclusive-write, and concurrent-read-concurrent-write. However, due to the low-level nature of the PRAM, even the simplest algorithms designed for it involve tedious implementation details. In order to abstract away these specifics, we usually consider the equivalent work-depth paradigm, which measures the performance of a parallel algorithm using two parameters – total work, which is the total running time needed given only one processor, and depth, which is the total parallel time given a maximal number of processors. Moreover, in this framework, the different variants of PRAM are equivalent up to poly-logarithmic factors in work and depth, and therefore, can also be abstracted.

Massively Parallel Computation (MPC) Model. The massively parallel computation (MPC) model [KSV10] is considered to be the modern model of parallel computation. In this thesis, we only consider the most stringent *fully scalable MPC* model, where the graph edges are partitioned across machines each of local memory n^δ for some arbitrary constant $\delta \in (0, 1)$. The computation proceeds in synchronous rounds, where within each round, each machine can perform unbounded local computation on its local memory. At the end of each round, each machine can send and receive messages with other machines, with the constraint that the total size of the sent and received messages cannot be larger than its local memory n^δ. After all rounds are finished, the machines then collectively output a solution to the problem we want to solve on the input graph. The goal is to finish the computation in a fewest possible rounds, and in a smallest possible total machine memory.

Note that the fully scalable MPC setting is especially difficult for graph problems, as a machine cannot even store the identities of all vertices.

It is known [KSV10, GSZ11b] that a PRAM algorithm of depth D and total work W can be simulated in the fully scalable MPC model with $O(D)$ rounds and $O(W)$ total memory.

Query (Sublinear Time) Model. We consider the standard query model for sublinear time algorithms on general graphs (see e.g. Chapter 10 of [Gol17]). This model allows two types of query

access to the graph: (i) degree queries: what is the degree of a vertex v? (ii) neighbor queries: what is the i-th neighbor of a vertex v? The performance of an algorithms is measured by how many queries it makes to finish the computation. Ideally, we want an algorithm using $\ll n^2$ queries. Note that this is in general a highly nontrivial task, as even to determine whether a graph is connected, it necessarily requires $\Omega(n^2)$ queries [Gol17].

1.2. Our Results

We now introduce the problems we study in this thesis and the technical results we obtain for each of them within the context of prior work. Our results are as follows:

1. An improved maximum matching algorithm in the streaming model [ABKL23] (see Section 1.2.1).
2. First set of algorithms and lower bounds for weighted graph sparsification in turnstile streams [CKL22b] (see Section 1.2.2).
3. An approximate maximum flow algorithm in PRAM with near-linear work and poly-logarithmic depth [AKL$^+$24] (see Section 1.2.3).
4. Optimal algorithms and lower bounds for sublinear time hierarchical clustering [AKLP22a] (see Section 1.2.4).

Remark 1.2.1. *By standard simulation, our PRAM maximum flow algorithm also implies a fully scalable MPC algorithm with poly-logarithmic rounds and near-linear total memory (cf. Result 8).*

Remark 1.2.2. *As one of the applications of our parallel maximum flow algorithm, we also obtain near-optimal PRAM and fully scalable MPC algorithms for hierarchical clustering (cf. Section 1.2.3.1).*

1.2.1. An Improved Maximum Matching Algorithm in the Streaming Model

Given a graph $G = (V, E)$, a matching M in G is any collection of edges that share no endpoints. Finding maximum matchings has been a cornerstone of algorithm design starting from the work of [Kön16] over a century ago. Nevertheless, many fundamental questions regarding the complexity of this problem have remained unresolved, specifically in modern models of computations such as streaming graphs. Indeed, in the streaming model, despite significant attention, there has been no

improvement in certain key cases over longstanding barriers that have remained in place since the introduction of the model itself.

As we introduced above, in the streaming model, edges of an n-vertex graph appear one by one in a stream in an arbitrary order. The algorithm can read the edges in the arrival order while using a limited memory smaller than the input size, and output the solution at the end of the stream. The holy grail of algorithms here is one that uses $O(n \cdot \text{polylog}\,(n))$ memory and a single pass over the stream. The study of graph streaming algorithms were initiated by [FKM$^+$05] who already observed that there is a straightforward 1/2-approximation algorithm for matching in $O(n \log n)$ space[1]: greedily maintain a maximal matching in the stream. They further proved that finding an exact maximum matching requires $\Omega(n^2)$ space, which matches the trivial algorithm that stores the entire input via its adjacency matrix.

Almost two decades since [FKM$^+$05], there are still no better algorithms for matchings than these two straightforward solutions. On the lower bound front, a series of work by [GKK12] and [Kap13] culminated in a recent work of [Kap21] that rules out better than $1/(1+\ln 2) \approx 0.59$ approximation in $n^{1+o(1/\log\log n)}$ space. This lack of progress has led researchers to consider various relaxations of the problem, in particular by allowing a few more passes over the input (e.g., in [KMM12, KT17, Kon18, KN21, Ass22, FS21]) or assuming random arrival of edges in the stream (e.g., in [KMM12, ABB$^+$19, FHM$^+$20, Ber20, AB21])[2]. At this point, beating 1/2-approximation factor of the greedy algorithm in $O(n \cdot \text{polylog}\,(n))$ space, or even much larger than that, has become one of the most central open questions of the graph streaming literature; see, e.g., [KMM12, McG14, Kap21, Waj20, FS21] for various references to this question.

Our Contributions. We present the first algorithms that beat the aforementioned barriers for finding matchings in streaming graphs with non-trivial albeit quite small factors:

Result 1. *There is a randomized $(1-o(1))$-approximate matching algorithm in the streaming model using $n^2/(\log^* n)^{\Omega(1)}$ space and polynomial time.*

[1] Throughout, as is standard, we always measure the space of streaming algorithms in *bits*.

[2] See the papers of [FS21] and [AB21], respectively, for the state of the art in each case, and more details on previous work on each relaxation.

This is the first single-pass $o(n^2)$-space algorithm for matchings in adversarial-order streams with better than 1/2-approximation guarantee. Moreover, combined with the lower bound of $\Omega(n^2)$ space by [FKM+05] for computing exact matchings, Result 1 shows the first provable separation between the space complexity of computing nearly-optimal versus exact-optimal matchings in single-pass streams.

The key idea behind our result is to maintain a matching sparsifier that we call *matching cover*—introduced by [GKK12] in spirit of cut/spectral sparsifiers—that is a "sparse" subgraph which approximately preserve matchings in each induced subgraph of the input graph. We present a polynomial time algorithm for constructing $o(n^2)$-size matching covers using Szemerédi's Regularity Lemma [Sze75] and along the way extend them to general graphs ([GKK12] only proves their existence and for bipartite graphs). We then show this new construction can be maintained in streaming graphs using several new ideas combined with standard tools from prior work specific to the model.

1.2.2. First Algorithms and Lower Bounds for Weighted Graph Sparsification in the Linear Sketching and Turnstile Streaming Models

Graph sparsification is a process that reduces the number of edges in a dense graph significantly while preserving certain useful properties. Besides being an interesting problem in its own right, graph sparsification has also been used as a fundamental building block in many modern graph algorithms such as maximum flow and minimum cut algorithms [BK15, She13a, KLOS14, Pen16], solvers for graph structured linear systems [ST14a, CKM+14], and graph clustering [CSWZ16].

In addition to designing fast algorithms in the traditional computational model, a rich body of work has also studied graph sparsification by *linear sketching*. As we discussed above, in this setting, we can only access the input graph by taking *linear measurements*, each of which returns a linear combination of the edge weights, and the goal is then to compute a sparsifier of the input graph using as few measurements as possible. To state the previously known results in this setting, let us first recall the definitions of two extensively studied graph sparsifiers that we will study in this work.

Definition 1.2.3 (Cut sparsifiers)**.** Given a weighted graph $G = (V, E, w)$ and a parameter $\varepsilon \in (0, 1)$, another weighted graph $H = (V, F, w')$ with $F \subseteq E$ is called a $(1+\varepsilon)$-*cut sparsifier* of G if for every cut $(S, V - S)$, its weight $w_G(S, V - S)$ in G and its weight $w_H(S, V - S)$ in H satisfy that $(1-\varepsilon)w_G(S, V - S) \leqslant w_H(S, V - S) \leqslant (1+\varepsilon)w_G(S, V - S)$.

Definition 1.2.4 (Spectral sparsifiers)**.** Given a weighted graph $G = (V, E, w)$ with Laplacian matrix L_G and a parameter $\varepsilon \in (0, 1)$, another weighted graph $H = (V, F, w')$ with Laplacian matrix L_H and $F \subseteq E$ is called a $(1+\varepsilon)$-*spectral* sparsifier of G if for every vector $x \in \mathbb{R}^n$ we have $(1-\varepsilon)x^T L_G x \leqslant x^T L_H x \leqslant (1+\varepsilon)x^T L_G x$.

A seminal work by Ahn, Guha, and McGregor [AGM12b] showed that one can compute a $(1+\varepsilon)$-cut sparsifier of an unweighted graph using $\widetilde{O}(n\varepsilon^{-2})$ linear measurements, which is nearly optimal. Then subsequent works by Kapralov, Lee, Musco, Musco, and Sidford [KLM$^+$14] and Kapralov, Mousavifar, Musco, Musco, Nouri, Sidford, and Tardos [KMM$^+$20] showed that one can also compute a $(1+\varepsilon)$-spectral sparsifier of an unweighted graph using $\widetilde{O}(n\varepsilon^{-2})$ linear measurements.

In all of these works [AGM12b, KLM$^+$14, KMM$^+$20], the authors also showed that their linear sketching algorithms can also be applied to computing graph sparsifiers of weighted graphs in *dynamic streams*, where the input graph is given by a stream of *insertions and deletions of weighted edges*, and the goal is to compute a sparsifier of the input graph using a small amount of space. In all these works, this is achieved by grouping the edge weights geometrically, and then applying the linear sketching algorithm for *unweighted graphs* to the subgraph induced by edges in each weight group. Note that this approach crucially requires that if an edge e is inserted with weight w_e, then any subsequent deletion of the edge e again reveals its weight and it must be *identical* to w_e. This is crucial to ensuring that each edge e is inserted into or deleted from the same geometric group.

However, the operation of grouping edges by weight is not linear, and as a result, the above approach for extending unweighted sketches to weighted ones is not implementable in the more general *turnstile streams*, where the graph is given as a stream of *arbitrary edge weight updates*. Surprisingly, little seems to be known about graph sparsification in this setting, or in the closely related linear sketching setting with input graphs being arbitrarily weighted.

Our Contributions. We initiate the study of *weighted* graph sparsification by linear sketching and in the turnstile streaming model. To state our results, we need to introduce some notation first. Let $w \in \mathbb{R}^{\binom{n}{2}}_{\geqslant 0}$ denote the weights of the edges of the input graph, where $w_e = 0$ means that there is no edge in the edge slot e. As we discussed above, a *linear sketch of $N = N(n)$ measurements* consists of a (random) *sketching matrix* $\Phi \in \mathbb{R}^{N \times \binom{n}{2}}$, and a (randomized) *recovery algorithm* $\mathcal{A}$ that takes as input $\Phi w \in \mathbb{R}^N$ and outputs a sparsifier of the graph with edge weights w. Note that, by definition, the linear sketch is *non-adaptive*.

We focus on a natural class of linear sketches that we call *incidence sketches*, in which each row of the sketching matrix Φ is supported[3] on edges incident on a *single* vertex (which could be different for different rows). This class captures all linear sketches that are implementable in a distributed computing setting, where the edges are stored across n machines such that machine i has all edges incident on the i^{th} vertex (a.k.a.*simultaneous communication model*). Moreover, it also covers *all* aforementioned linear sketches used in previous works for unweighted cut sparsification [AGM12b] and spectral sparsification [KLM+17, KMM+20].

We now present our results for computing the two kinds of sparsifiers in weighted graphs. When describing our results, we use $w_{\max}$ and $w_{\min}$ to denote the largest and the smallest non-zero edge weights, respectively, and always assume $w_{\max} \geqslant 1 \geqslant w_{\min}$. We also write $\widetilde{O}(\cdot)$ to hide $\text{polylog}(n, \varepsilon^{-1}, \frac{w_{\max}}{w_{\min}})$ factors.

Weighted Cut Sparsification. We design an incidence sketch with a near-linear number of measurements for computing a $(1+\varepsilon)$-cut sparsifier of a weighted graph.

Result 2. *For any $\varepsilon \in (0,1)$, there exists an incidence sketch with random sketching matrix $\Phi_1 \in \mathbb{R}^{N_1 \times \binom{n}{2}}$ satisfying $N_1 \leqslant \widetilde{O}(n\varepsilon^{-3})$ and a recovery algorithm $\mathcal{A}_1$, such that for any $w \in \mathbb{R}^{\binom{n}{2}}_{\geqslant 0}$, $\mathcal{A}_1(\Phi_1 w)$ returns, with probability $1 - \frac{1}{\text{poly}(n)}$, a $(1+\varepsilon)$-cut sparsifier of the graph with edge weights w.*

Thus, we achieve a similar performance as the linear sketch for unweighted graphs as in [AGM12b], which uses $O(n\varepsilon^{-2}\,\text{polylog}(n))$ measurements. It is well known that even to detect the connectivity

[3]Recall that the support of a vector is the set of indices at which it is non-zero.

of a graph, $\Omega(n)$ linear measurements are needed. Therefore, our upper bound in Result 2 is nearly optimal in n.

Similar to [Ind06, AGM12b, KLM+14, KMM+20], we can turn our linear sketching algorithm to a low space streaming algorithm.

Result 3 (Corollary of Result 2)**.** *There is a single pass turnstile streaming algorithm with $\widetilde{O}(n\varepsilon^{-3})$ space that, at any given point of the stream, recovers a $(1+\varepsilon)$-cut sparsifier of the current graph with high probability.*

Once again, note that in the turnstile model, the stream consists of arbitrary edge weight updates.

Weighted Spectral Sparsification. We design an incidence sketch with about $n^{6/5}$ measurements for computing a $(1+\varepsilon)$-spectral sparsifier of a weighted graph.

Result 4. *For any $\varepsilon \in (0,1)$, there exists an incidence sketch with random sketching matrix $\Phi_2 \in \mathbb{R}^{N_2 \times \binom{n}{2}}$ satisfying $N_2 \leqslant \widetilde{O}(n^{6/5}\varepsilon^{-4})$ and a recovery algorithm $\mathcal{A}_2$, such that for any $w \in \mathbb{R}^{\binom{n}{2}}_{\geqslant 0}$, $\mathcal{A}_2(\Phi_2 w)$ returns, with probability $1-\frac{1}{\mathrm{poly}(n)}$, a $(1+\varepsilon)$-spectral sparsifier of the graph with edge weights w.*

Similar to the cut sparsification case, we have the following corollary:

Result 5 (Corollary of Result 4)**.** *There is a single pass turnstile streaming algorithm with $\widetilde{O}(n^{6/5}\varepsilon^{-4})$ space that, at any given point of the stream, recovers a $(1+\varepsilon)$-spectral sparsifier of the current graph with high probability.*

We complement this result by showing that a superlinear number of measurements are indeed necessary for any incidence sketch to recover some $O(1)$-spectral sparsifier.

Result 6. *There exist constants $\varepsilon, \delta \in (0,1)$ such that any incidence sketch of N measurements that computes a $(1+\varepsilon)$-spectral sparsifier with probability $\geqslant 1-\delta$ on any w must satisfy $N \geqslant n^{21/20-o(1)}$.*

Note that this is in sharp contrast to the unweighted case, where a near-linear number of incidence sketch measurements are sufficient for computing an $O(1)$-spectral sparsifier [KLM+17, KMM+20].

Result 6 also draws a distinction between spectral sparsification and cut sparsification, as for the latter a near-linear number of measurements are enough even in the weighted case (by Result 2).

1.2.3. Parallel Approximate Maximum Flow Algorithms Achieving Simultaneously Near-Linear Work and Poly-logarithmic Depth

The *maximum flow* problem, or equivalently, the *minimum congestion flow* problem is one of the oldest and most well-studied combinatorial optimization problems that finds numerous applications across computer science and engineering. Formally, we are given a directed, capacitated flow network (graph) $G = (V, E, c)$ with $|V| = n$ vertices and $|E| = m$ edges with positive edge capacities $c \in \mathbb{R}^E$, along with a set of vertex demands $b \in \mathbb{R}^V$ specifying the desired excess flow at each vertex where $\sum_{v \in V} b_v = 0$. The objective is to find a flow $f \in \mathbb{R}^E$ that satisfies demands b, while minimizing the maximum congestion $|f_e|/c_e$ on any edge $e \in E$ in the network, i.e.

$$\min \ \|C^{-1}f\|_\infty \ \text{ s.t. } Bf = b \ \textit{(flow conservation constraints)}; \ f \geqslant 0, \tag{1.1}$$

where $C \in \mathbb{R}^{E \times E}$ is a diagonal matrix of edge capacities with $C_{e,e} = c_e$, and $B \in \{-1, 0, 1\}^{V \times E}$ is the vertex-edge incidence matrix with $B_{v,e}$ being 1 if $e = (u, v)$, -1 if $e = (v, u)$ and 0 otherwise. For the special case where the edges are undirected, the formulation is identical except that the edges are oriented arbitrarily, and the non-negativity constraint on the flows is dropped; the sign of the flow on an edge specifies its direction relative to the edge orientation.

The maximum flow problem has a rich history, starting with the work of [Dan51] and [FF56], who first proposed algorithms with pseudo-polynomial running times of $O(mn^2U)$ and $O(m^2U)$ respectively, for networks with maximum edge capacity U. Since then, substantial effort has been devoted to designing increasingly efficient algorithms for this problem. This has resulted in a sequence of exciting developments, with initial improvements coming from primarily combinatorial ideas such as shortest augmenting paths and blocking flows [Din70, EK72, Kar73, Din73, ET75, BK04], push-relabeling [GT88, Gol08, OG21], pseudo-flows [Hoc08, CH09], and capacity scaling [AO95, GR98], eventually culminating in the recent breakthrough $O(m^{1+o(1)})$ time result of [CKL$^+$22a] achieved through a combination of second-order continuous optimization techniques (interior point methods)

and efficient dynamic graph data-structures. Meanwhile, for the weaker objective of finding an approximately maximum flow in undirected graphs, even faster algorithms have been developed using simpler first-order continuous optimization techniques [She13b, Pen16, She17a, She17b]. In particular, [Pen16] showed that in undirected graphs, a $(1-\varepsilon)$-approximate maximum flow can be computed in just $O(m \text{ poly}(1/\varepsilon, \log n))$ time.

However, these aforementioned algorithmic results were largely developed in the sequential setting, and are not readily parallelizable. Moreover, the question of designing fast *parallel* algorithms for max-flows has remained surprisingly under-explored.

For the question of designing parallel algorithms for max flows in the PRAM model, the literature is sparse. Recall that in the PRAM model, we measure the performance of a parallel algorithm using two parameters – total work, which is the total running time needed given only one processor, and depth, which is the total parallel time given a maximal number of processors. For finding exact max flows, early results of [SV82, Ram90] achieved $\tilde{O}(n^2)$ depth and $\tilde{O}(mn)$ work. More recently, [PF22] improved the depth to $\tilde{O}(n)$ while retaining the same asymptotic work. For the weaker objective of finding an approximate max flow, the combined results of [Mad16] and [PS14] imply an algorithm with $\tilde{O}(\sqrt{m})$ depth and $\tilde{O}(m^{1.5})$ work at the cost of a $O(1/\text{poly}(n))$ additive error in the flow value. A much earlier work of [SS91] also implies a $O(\text{poly}(\log n, \log(1/\varepsilon)))$ depth algorithm for finding a $(1-\varepsilon)$-approximate max flow via a reduction to finding maximum matchings in bipartite graphs. However, this comes at the expense of an unspecified polynomial blow-up in work, a common issue with many of the early results for parallel algorithms. The question of whether it is possible to *simultaneously* have a poly-logarithmic depth and near-linear work approximate max-flow algorithm remained open.

Our Contributions. In this work, we answer this in the affirmative for undirected graphs. Namely, we show the following main result.

Result 7. *There is a randomized PRAM algorithm that given an undirected capacitated graph $G = (V, E, c)$, $s, t \in V$, and precision $\varepsilon > 0$, computes both a $(1-\varepsilon)$-approximate s-t maximum flow and a $(1+\varepsilon)$-approximate s-t minimum cut with high probability in $O\left(\varepsilon^{-3} \operatorname{polylog} n\right)$ depth and $O\left(m\varepsilon^{-3} \operatorname{polylog} n\right)$ total work.*

We remark that our parallel maximum flow algorithms are obtained by a novel parallel construction of a flow sparsifier that in turn guides us in routing a near-maximum flow.

As we discussed above, a more recent model of parallel computation is the *massively parallel computation* (MPC) model, which is a common abstraction of many MapReduce-style computational frameworks (see i.e. [BKS17, ANOY14, GSZ11b]). In this model, the input data is partitioned across multiple machines that are connected together through a communication network. The computation proceeds in synchronous rounds where in each round, the machines can perform unlimited local computation on their local memory, but cannot communicate with other machines. Between rounds, machines can communicate, so long as the total size of messages sent and received by any machine does not exceed the size of its local memory. The performance of an MPC algorithm is measured by the number of rounds needed to complete the computation, the size of the local machine memory, as well as the total memory used across all machines. A simulation result of [KSV10, GSZ11b] shows that any PRAM algorithm with D depth and W work can be simulated by an MPC algorithm with $O(D)$ rounds and $O(W)$ total memory even in the most stringent *fully-scalable* regime[4] where the local memory size is $O(n^\delta)$ for any constant $\delta > 0$. As we noted above, the fully scalable MPC regime is especially hard for graph problems as a machine cannot even store the identities of all vertices. Nonetheless, using the simulation result in [KSV10, GSZ11b], we obtain the following corollary of our main result.

[4]The dependence of the the local memory parameter (the constant $\delta > 0$) on the number of rounds and total memory is a multiplicative $O(1/\delta)$, and is usually omitted.

Result 8 (Corollary of Result 7)**.** *There is a randomized MPC algorithm that given an undirected capacitated graph $G = (V, E, c)$, $s, t \in V$ and precision $\varepsilon > 0$, computes both a $(1-\varepsilon)$-approximate s-t maximum flow and a $(1+\varepsilon)$-approximate s-t minimum cut with high probability in $O\left(\varepsilon^{-3} \operatorname{polylog} n\right)$ rounds, $O\left(m\varepsilon^{-3} \operatorname{polylog} n\right)$ total memory and $O\left(n^{\delta}\right)$ local memory, for any constant $\delta > 0$.*

1.2.3.1 Applications of Our Parallel Approximate Maximum Flow Algorithm

Our result for approximate max flows has broad implications to other related graph problems, where it implies either new or substantially improved parallel algorithms (both PRAM and MPC) for them. The results in this section are formalized in Section 5.6.

Sparsest cut and balanced sparsest cut: The *sparsest cut* problem is an important subroutine for divide-and-conquer based approaches for several graph problems and has many applications including image segmentation, VLSI design, clustering and expander decomposition. In this problem, given a weighted undirected graph $G = (V, E, c)$, the objective is to find a cut $(S, \bar{S})$ with minimum sparsity $\phi(S)$ which is defined as $\phi(S) := c(E(S, \bar{S}))/\min\{|S|, |\bar{S}|\}$, where $c(E(S, \bar{S}))$ is the total weight of the edges going across the cut. The *balanced sparsest cut* problem is a variant of this problem where there is an additional requirement that the cut $(S, \bar{S})$ must be β-balanced, i.e. $\min\{|S|, |\bar{S}|\} \geqslant \beta n$ for some given parameter $\beta \in (0, 1/2)$. Balanced sparse cuts are useful in applications where one wants the divide-and-conquer tree to have low-depth.

The sparsest cut problem is NP-hard and the best known approximation factor of $O(\sqrt{\log n})$ is achieved by an SDP-based algorithm due to Arora, Rao and Vazirani [ARV09b]. However, this algorithm is highly sequential, and computationally expensive. The most efficient algorithms for sparsest cut problem are based on the cut-matching game framework of Khandekar, Rao and Vazirani [KRV06], which effectively reduces the sparsest cut problem to a poly-logarithmic number of single commodity max-flow computations. In this framework, a cut-player and a matching-player play an alternating game; in each round, the former produces a bisection $(S, \bar{S})$ of vertices, and the latter produces a perfect matching between S and $\bar{S}$ that can be embedded in the underlying graph. The game ends when either the cut player produces a bisection for which the matching

player cannot find a perfect matching (i.e. a sparse cut has been found), or the union of the perfect matchings produced thus far form an expander (i.e. an expander can be embedded in the underlying graph, certifying its expansion). [KRV06] showed that there is a cut strategy with runtime $T_{\text{cut}} = \tilde{O}(n)$ such that this game terminates in $\alpha = O(\log^2 n)$ rounds, regardless of the matching player's strategy. Furthermore, the matching player's strategy can be implemented using a max-flow computation with the sources and sinks being the two sides of the bipartition. In combination, this framework produces an $O(\alpha)$-approximation to sparsest cut with a runtime of $O(\alpha \cdot (T_{\text{cut}} + T_{\text{flow}}))$, where α is an upper bound on the number of rounds of this game, T_{cut} is the time to implement cut player's strategy and T_{flow} is the time to compute a single commodity max-flow. This framework also generalizes to the problem of β-balanced sparsest cut for any constant β with the same approximation factor and running time.

Nanongkai and Saranurak [NS17] further showed that the matching player's strategy can be implemented using approximate max-flow computations rather than exact max-flows. Building on [NS17], we show that our parallel algorithm for max-flows (Result 7) can be used to implement the matching player's strategy, yielding a $O(\text{polylog}\, n)$ depth and $\tilde{O}(m)$ work algorithm for the matching player. Furthermore, we show that the cut player's strategy can also be implemented in $O(\text{polylog}\, n)$ depth and $\tilde{O}(m)$ work using our new *parallel flow decomposition* algorithm (see Section 5.1.2 for a discussion). In combination, this gives a $O(\text{polylog}\, n)$ depth and $\tilde{O}(m)$ work algorithm for sparsest cuts that has an approximation factor of $O(\log^3 n)$. These results also imply a bicriteria approximation algorithm for β-balanced sparsest cut with the same depth and work. Specifically, our algorithm finds a $(\beta/\log^2 n)$-balanced cut whose sparsity is at most $O(\log^3 n)$ times the sparsity of an optimal β-balanced cut.

Minimum cost hierarchical clustering: Before presenting our optimal sublinear time algorithms and lower bounds results for the hierarchical clustering problem in Section 1.2.4, we first discuss our near-optimal parallel algorithms for it, obtained as an application of our parallel maximum flow algorithm. Hierarchical clustering is a fundamental problem in data analysis where the goal is to recursively partition data into clusters which results in a rooted tree structure. This problem has wide-ranging applications across different domains including phylogenetics, social network analysis,

and information retrieval. While the study of hierarchical clustering goes back several decades, Dasgupta [Das16] initiated its study from an optimization viewpoint. Specifically, he proposed a minimization objective for hierarchical clustering on similarity graphs that measures the cost of a hierarchy as the sum of costs of its internal splits, which in turn is measured as the total edge-weight across the split scaled by the size of the cluster at that split. [Das16] and subsequent work by [CKMM18, CC17] showed that the hierarchical clustering produced by recursively splitting the graph using a α-approximate sparsest cut subroutine results in a hierarchy that is an $O(\alpha)$-approximation for this minimization objective. Furthermore, using an α-approximate β-balanced sparsest cut subroutine results in the same approximation with an additional property that the depth of the tree is $O(\log n)$ for $\beta = \Omega(1)$. [CC17] also showed that a (α, β')-bicriteria approximation oracle for β-balanced min-cut is sufficient to achieve a $O(\alpha/\beta')$-approximation for minimum cost hierarchical clustering. As a consequence, using our parallel balanced min-cut algorithm as a subroutine, we get the first parallel algorithm that computes a tree that is a $O(\log^5 n)$-approximation to Dasgupta's objective with $\tilde{O}(m)$ work and $O(\text{polylog}\, n)$ depth. We refer the reader to Section 5.6.2 for a formalization of this discussion.

Fair cuts and approximate Gomory-Hu trees: Li *et al.* [LNPS23] recently introduced the notion of fair cuts for undirected graphs which are a "robust" generalization of approximate min-cuts. Specifically, for $\alpha \geqslant 1$, a s-t-cut is α-*fair* if there exists an s-t flow that uses at least $1/\alpha$-fraction of the capacity of every edge in the cut. [LNPS23] showed that a near-linear time oracle for computing a fair cut is useful for obtaining near-linear time algorithms for several applications including computation of (approximate) all-pairs maxflow values (using approximate Gomory-Hu trees). They also showed that, given an unweighted graph, a fair cut can be computed in parallel using $m^{1+o(1)}$ work and $n^{o(1)}$ depth, implying a similar result for approximate Gomory-Hu trees. Our results improve upon their results by giving parallel algorithms for fair cuts and approximate Gomory-Hu trees that have $O(m\ \text{polylog}\, n)$ work and $O(\text{polylog}\, n)$ depth. The key technical tool that results in this improvement is our $O(\text{polylog}\, n)$ depth construction of a $O(\text{polylog}\, n)$-*congestion approximator* (see Section 5.1.1 for a discussion).

1.2.4. Optimal Algorithms and Lower Bounds for Sublinear Time Hierarchical Clustering

Recall that in Section 1.2.3.1, we discussed our near-optimal parallel algorithms for hierarchical clustering, obtained as an application of our parallel maximum flow algorithm. In this section, we introduce our optimal algorithms and lower bounds for the hierarchical clustering problem in the sublinear time setting. We start with a more thorough introduction of the problem.

Hierarchical clustering (HC) is a popular unsupervised learning method for organizing data into a dendrogram (rooted tree). It can be viewed as clustering datapoints at multiple levels of granularity simultaneously, with each leaf of the tree corresponding to a datapoint and each internal node of the tree corresponding to a cluster consisting of its descendent leaves. Much of the technical development of HC originated in the field of *phylogenetics*, where the motivation was to organize the different species into an evolutionary tree based on genomic similarities [ESBB98]. Since then, this tool has seen widespread use in data analysis for a variety of domains ranging from *social networks*, *information retrieval*, *financial markets* [GSZ$^+$11a, Ber06, SMR08] amongst many others.

Due to its popularity, HC has been extensively studied and several algorithms have been proposed. The most prominent amongst these are *bottom-up agglomerative* algorithms such as average linkage, single linkage, complete linkage etc. (see Chapter 14 in [HTFF09]). However, despite these advances on the algorithmic front, very few formal guarantees were known for their performance, primarily owing to a historic lack of a well defined objective function. Therefore, the study of HC was largely empirical in nature for a long time.

A part of this issue was recently resolved, when [Das16] proposed an objective function for similarity-based HC. This has since sparked interest in both the theoretical computer science as well as machine learning communities, for designing algorithms with provable guarantees for this objective [CC17, CNC18, CCN19, CKMM19, AAV20]. The formal description is as follows: given as input a weighted undirected graph $G = (V, E, w)$ with n vertices (datapoints) and m edges with positive edge weights corresponding to pairwise similarities between its endpoints, the objective is to find a

hierarchy $\mathcal{T}$ over leaf nodes corresponding to the vertices V that minimizes the cost function

$$\mathsf{cost}_G(\mathcal{T}) := \sum_{\{i,j\} \in E} w_{ij} \cdot |\mathcal{T}_{ij}|\,, \tag{1.2}$$

where $\mathcal{T}_{ij}$ is the subtree rooted at the least-common ancestor of i, j in $\mathcal{T}$ and $|\mathcal{T}_{ij}|$ is the number of descendent leaves in $\mathcal{T}_{ij}$. Intuitively, $\mathsf{cost}_G(\mathcal{T})$ incentivizes cutting *heavy* edges at lower levels in $\mathcal{T}$, thereby placing more similar datapoints closer together. This objective has been shown to have several desirable properties, including one that guarantees an optimal tree which is binary [Das16].

This minimization objective however, turns out to be NP-hard. Consequently, [Das16] and other subsequent work explored this objective from an approximation algorithms perspective [RP17, CC17, CNC18, CCN19, CKMM19, AAV20]. The best known polynomial time approximation is $O(\sqrt{\log n})$ which is achieved by the recursive sparsest cut (RSC) algorithm [Das16, CC17, CKMM19]. It is also known that no constant factor polynomial time approximation is possible for this objective under the *small-set expansion* (SSE) hypothesis [CC17].

In this work, we study the above minimization objective for HC in the context of *massive* graphs. While the currently known best algorithm can be considered "efficient" in the classical sense, i.e. requires polynomial time, this complexity can be prohibitive in many modern applications of HC that deal with staggering volumes of data. For example, current social networks contain billions of edges which imposes serious limits on their processing. Therefore, an alternative model of computation needs to be considered in the context of such massive graphs. In this work, we consider the *general graph (query) model* [Gol17] for time efficiency, where the edges can be accessed via degree and neighbor queries. The focus of our work is the following fundamental question:

Can we solve hierarchical clustering faster than reading the entire data?

To appreciate the difficulty in designing sublinear time algorithms for this question, note that even to decide whether a graph is connected one necessarily needs $\Omega(n^2)$ queries [Gol17].

Our Contributions. We provide an *almost complete resolution* to this question by providing *matching upper and lower bounds* for sublinear time algorithms in the query model. Before presenting our result, we set up some notation first.

Table 1.1: **Summary of Results.** Each row gives an upper and lower bound on the number of queries required for $\widetilde{O}(1)$ approximation in the query model.

	Graph Density	**Upper Bound**	**Lower Bound**
Query Model	$1 < \zeta \leqslant 4/3$	$\widetilde{O}(n^{\zeta})$	$\Omega(n^{\zeta - o(1)})$
(Sublinear Time)	$4/3 < \zeta \leqslant 3/2$	$\widetilde{O}(n^{4-2\zeta})$	$\Omega(n^{4-2\zeta-o(1)})$
Edges $m = \Theta(n^{\zeta})$ in G	$3/2 \leqslant \zeta < 2$	$\widetilde{O}(n)$	$\Omega(n)$

Notation. We use n and m to refer to the number of vertices and edges in the input graph, respectively. We use $\widetilde{O}(\cdot)$ to suppress multiplicative $O(\log^c n)$ factors for constant c, and the term "w.h.p." implies with probability $1 - 1/\text{poly}(n)$.

For illustration purpose, here we present our results for unweighted graphs only. However these results can be readily extended to weighted graphs by standard weight grouping tricks; we defer the discussion of the weighted extension to the main body of the thesis.

We summarize our upper bound and lower bound results in Table 1.1.

Our first result is a sublinear time algorithm for hierarchical clustering:

Result 9. *There exists an algorithm that given query access to any unweighted graph G with $m = \Theta(n^{\zeta})$ for $\zeta \in [0, 2]$, can find a $(1 + o(1)) \cdot \phi$-approximate HC of G w.h.p. using $\widetilde{O}(g(n, \zeta))$ queries, where $g(n, \zeta) \leqslant n^{4/3}$ is given by $g(n, \zeta) = \max\{n, n^{\zeta}\}$ when $\zeta \in [0, 4/3]$, $g(n, \zeta) = \max\{n, n^{4-2\zeta}\}$ when $\zeta \in (4/3, 2]$. Moreover, given any (arbitrarily small) constant $\tau > 0$, the algorithm can find an $O(\sqrt{\log n})$-approximate HC of G w.h.p. in $\widetilde{O}(g(n, \zeta) + n^{1+\tau})$ time.*

It is interesting to observe that the query complexity $g(n, \zeta)$ reduces as the graph becomes denser. This is due to the internal mechanism of our sublinear time algorithm. Specifically, our algorithm works by constructing a relaxed notion of cut sparsifier of the input graph, and then running a hierarchical clustering algorithm on the resulting sparsifier. Our notion of cut sparsifiers is relaxed in the sense that we allow an additive error in the cut approximation in addition to the usual multiplicative approximation.

Our key observation is that as the density of the graph increases, the cost of the optimal HC also increases, which allows us to tolerate a larger additive error in our cut sparsifier, thereby making it

sparser. Consequently, we get faster algorithms for denser graphs.

Complementing our sublinear time algorithm, we also prove a matching lower bound for *every graph density*.

Result 10. *Let $\zeta \in [0,2]$ be any constant. Let $\mathcal{A}$ be a randomized algorithm that, on any input unweighted graph with $\Theta(n^{\zeta})$ edges, outputs with high probability a* $\text{polylog}(n)$*-approximate HC. Then $\mathcal{A}$ necessarily uses at least $\Omega(g(n,\zeta))$ queries, where we define $g(n,\zeta) = \max\{n, n^{\zeta - o(1)}\}$ when $\zeta \in [0,4/3]$, $g(n,\zeta) = \max\{n, n^{4-2\zeta-o(1)}\}$ when $\zeta \in (4/3,2)$, and $g(n,\zeta) = 0$ when $\zeta = 2$.*

1.3. Roadmap

In CHAPTER 2, we introduce the basic background knowledge and tools needed for the remaining part of the thesis. In CHAPTER 3, we present our improved streaming algorithm for the maximum matching problem. In CHAPTER 4, we present the first set of results for weighted graph sparsification in the turnstile streaming and linear sketching models. In CHAPTER 5, we present our PRAM algorithm for the approximate maximum flow problem achieving near-linear work and poly-logarithmic depth. In CHAPTER 6, we present our optimal sublinear time algorithms and lower bounds for the hierarchical clustering problem.

CHAPTER 2

BACKGROUND

2.1. Notations

For any integer $t \geqslant s \geqslant 1$, we let $[t] := \{1, \ldots, t\}$ and let $[s, t] = \{s, \ldots, t\}$. We use the term with high probability, abbreviated w.h.p., to imply probability at least $1 - 1/n^c$ for any desirably large constant $c \geqslant 1$ (that might affect the hidden constants in our statements).

For a graph $G = (V, E)$, we use $V(G) = V$ to denote the set of vertices and $E(G) = E$ to denote the edges. For any subsets of edges $F \subseteq E$ and disjoint subsets of vertices $X, Y \subseteq V$, we use $X(F)$ and $Y(F)$ to denote the edges of F incident on X and Y, respectively, and $F(X, Y)$ to denote the edges of F going between X and Y. Similarly, we use $G[X]$ and $G[X, Y]$ to respectively denote the subgraph of G induced on vertices X, and the bipartite subgraph of G between vertices X and Y. For any $p \in [0, 1]$, we use $G[p]$ to denote a random subgraph of G that includes each edge of G independently with probability p.

2.2. Matchings and Graph Combinatorics

For any graph G, $\mu(G)$ denotes the size of the maximum matching in G. We have,

Fact 2.2.1. *Any graph G has at most $2n \cdot \mu(G)$ edges.*

The proof of Fact 2.2.1 is simply based on picking an arbitrary edge of the graph and adding to a matching, removing at most $2n$ edges incident on this edge, and repeating until the graph is empty.

We will also need the following version of Hall's theorem.

Proposition 2.2.2 (Extended Hall's marriage theorem; cf. [Hal87])**.** *Let $G = (L, R, E)$ be any bipartite graph with $|L| = |R| = n$. Then $\max(|A| - |N_G(A)|) = n - \mu(G)$, where A ranges over all subsets of L and R, and $N_G(A)$ denotes the neighbors of A in G.*

2.2.1. Szemerédi's Regularity Lemma

Szemerédi's Regularity Lemma [Sze75] is a powerful tool in extremal combinatorics. Loosely speaking, it says that every dense graph can be well-approximated by a "small" collection of random-like

subgraphs. To formally state the lemma, we need a few definitions.

Let $G = (V, E)$ be any given graph, and $A, B \subseteq V$ be any disjoint vertex subsets. We write $e(A, B)$ to denote the number of edges between A, B. If $A, B \neq \emptyset$, we define the **density** of edges between A and B by:

$$d(A, B) := \frac{e(A, B)}{|A||B|}.$$

For a parameter $\gamma \in (0, 1)$, we say (A, B) is **γ-regular** if for every $X \subseteq A$ and $Y \subseteq B$ satisfying $|X| \geqslant \gamma \cdot |A|$ and $|Y| \geqslant \gamma \cdot |B|$, we have $|d(A, B) - d(X, Y)| < \gamma$.

Let $C_0, C_1, \ldots, C_k$ be a partition of the vertex set V. We say this partition is **equitable** if the classes $C_1, \ldots, C_k$ all have the same size. We will call C_0 the **exceptional class**. We say this partition is γ-regular if both of the following statements are true:

1. It is equitable and $|C_0| \leqslant \gamma n$.
2. All but at most $\gamma\binom{k}{2}$ of the pairs C_i, C_j for $1 \leqslant i < j \leqslant k$ are γ-regular.

Instead of the original formulation of Szemerédi's Regularity Lemma in [Sze75], we state an algorithmic version of it due to [ADL$^+$92].

Proposition 2.2.3 ([ADL$^+$92])**.** *There exists a function $Q : \mathbb{R}^+ \times \mathbb{R}^+ \to \mathbb{R}^+$ satisfying $\log^* Q(x, y) \leqslant \text{poly}(x, y)$ for all x, y, such that, given any n-vertex graph $G = (V, E)$ and $\gamma \in (0, 1), t \geqslant 1$, one can find in $n^\omega \cdot Q(t, 1/\gamma)$ time a γ-regular partition of V into $k + 1$ classes such that $t \leqslant k \leqslant Q(t, 1/\gamma)$.*

The algorithm in Proposition 2.2.3 can also be implemented in a space-efficient manner (which is needed for our streaming algorithms). See Section A.1 for a proof sketch.

Proposition 2.2.4. *Given query access to the adjacency matrix, the algorithm in Proposition 2.2.3 can be implemented in $O(n \cdot Q(t, 1/\gamma) \log n)$ space and $\text{poly}(n, Q(t, 1/\gamma))$ time.*

2.2.2. Fox's Triangle Removal Lemma

Similar to the Regularity Lemma, the Triangle Removal Lemma is another highly useful tool in extremal combinatorics. While original proofs of this lemma were based on the regularity

lemma, [Fox11] presented a proof that bypasses regularity lemma and thus obtains stronger bounds. We will use this result also in one of our streaming algorithms.

Proposition 2.2.5 ([Fox11])**.** *There exists an absolute constant $b > 1$ such that the following is true. For any $\gamma \in (0,1)$ let $\delta := \delta(\gamma)$ be inverse of the tower of twos of height $b \cdot \log(1/\gamma)$, i.e., $\delta^{-1} = 2 \uparrow\uparrow b \cdot \log(1/\gamma)$. Then, any n-vertex graph with at most $\delta \cdot n^3$ triangles can be made triangle-free by removing at most $\gamma \cdot n^2$ edges.*

2.2.3. Succinct Dynamic Dictionaries

We need to use succinct dynamic dictionaries from prior work in [BM99, Pag01, RR03]. For concreteness, we use the construction of [RR03] although the other ones work as fine also for us.

Proposition 2.2.6 (c.f. [RR03])**.** *There exists a dynamic data structure $\mathbf{D}$ for maintaining a subset S of size at most s from a universe U of size u that supports the following operations:*

- $\mathbf{D}.\texttt{insert}(a)$*: Inserts an element $a \in U$ to the set S;*
- $\mathbf{D}.\texttt{member}(a)$*: Returns whether the given element $a \in U$ belongs to U or not;*

The data structure requires $(1+o(1)) \cdot \log \binom{u}{s}$ bits of space to store S and answers each query in $O(1)$ amortized expected time or $O(s)$ worst-case deterministic time.

2.3. Linear Algebra and High Dimensional Gaussian Distributions

2.3.1. Matrices

Definition 2.3.1 (Pseudoinverse)**.** Let A be an $n \times n$ symmetric matrix. Let $\lambda_1, \lambda_2, \ldots, \lambda_n$ be the eigenvalues of A and let $v_1, v_2, \ldots, v_n$ be orthonormal (i.e. unit and orthogonal) eigenvectors of A, then by the spectral theorem $A = \sum_{i=1}^{n} \lambda_i v_i v_i^T$. The pseudoinverse of A is then defined as

$$A^{\dagger} \stackrel{\text{def}}{=} \sum_{\lambda_i \neq 0} \frac{1}{\lambda_i} v_i v_i^T. \tag{2.1}$$

Definition 2.3.2 (Matrix partial ordering)**.** For two $n \times n$ symmetric matrices A, B, we write $A \preceq B$ if for any vector $x \in \mathbb{R}^n$ we have $x^T A x \leqslant x^T B x$ (i.e. $B - A$ is positive semi-definite).

Fact 2.3.3 (Properties of the matrix partial ordering)**.**

1. *If $A \preceq B$ and $C \preceq D$, then $A + C \preceq B + D$.*
2. *If $A \preceq B$ where $A, B \in \mathbb{R}^{n\times n}$, then for any W with n rows, $W^T AW \preceq W^T \boldsymbol{B} W$.*
3. *For two positive semidefinite matrices A, B that have the same null space, $A \preceq B$ implies that $B^\dagger \preceq A^\dagger$.*

Theorem 2.3.4 (Matrix chernoff bound [Tro12])**.** *Let $X_1, \ldots, X_m \in \mathbb{R}^{n\times n}$ be independent random positive semidefinite matrices such that $\lambda_{\max}(X_i) \leqslant R, \forall i$ and $\mathbb{E}\left[\sum_{i=1}^m X_i\right] = I_{n\times n}$ where $I_{n\times n}$ is the $n \times n$ identity matrix. Then with probability at least $1 - 2n \exp\left\{-\frac{\varepsilon^2}{3R}\right\}$*

$$(1-\varepsilon)I \preceq \sum_{i=1}^m X_i \preceq (1+\varepsilon)I. \tag{2.2}$$

Definition 2.3.5 (Leverage scores)**.** Let $a_1, \ldots, a_m \in \mathbb{R}^n$ and $A = \sum_{i=1}^m a_i a_i^T \in \mathbb{R}^{n\times n}$. The leverage score of a_i w.r.t. A is defined to be $\tau_i(A) = a_i^T A^\dagger a_i$.

Fact 2.3.6. *Let $a_1, \ldots, a_m \in \mathbb{R}^n$ and $A = \sum_{i=1}^m a_i a_i^T \in \mathbb{R}^{n\times n}$. Also let $B = (a_1, \ldots, a_m) \in \mathbb{R}^{n\times m}$, so we have $A = BB^T$. Let $b \in \mathbb{R}^n$ be a vector in the span of $a_1, \ldots, a_m$. Then we have*

$$\min_{Bx=b} \|x\|_2^2 = b^T (BB^T)^\dagger b.$$

2.3.2. Multivariate Gaussian distributions

Definition 2.3.7. Let $\mu \in \mathbb{R}^d$ be a vector and $\Sigma \in \mathbb{R}^{d\times d}$ be a matrix. We say a random vector $x \in \mathbb{R}^d$ follows a multivariate Gaussian distribution with mean μ and covariance matrix Σ, denoted by $\mathcal{N}(\mu, \Sigma)$, if

1. Each x_i distributes as $\mathcal{N}(\mu_i, \Sigma_{ii})$, a univariate Gaussian with mean μ_i and variance Σ_{ii}.
2. $\mathbb{E}\left[(x_i - \mu_i)(x_j - \mu_j)\right] = \Sigma_{ij}$ for all pairs i, j. Or equivalently,

$$\mathbb{E}\left[(x-\mu)(x-\mu)^T\right] = \Sigma. \tag{2.3}$$

Fact 2.3.8. *The covariance matrix Σ is symmetric and positive semi-definite.*

Fact 2.3.9. *Let $x_1, x_2 \in \mathbb{R}^d$ be independent random vectors such that $x_1 \sim \mathcal{N}(\mu_1, \Sigma_1)$ and $x_2 \sim \mathcal{N}(\mu_2, \Sigma_2)$. Then $x_1 + x_2 \sim \mathcal{N}(\mu_1 + \mu_2, \Sigma_1 + \Sigma_2)$.*

Theorem 2.3.10 (ℓ_1-distance between multivariate Gaussians with the same mean [DMR18])**.** *Let $\mu \in \mathbb{R}^d$ and let $\Sigma_1, \Sigma_2 \in \mathbb{R}^{d \times d}$ be positive semidefinite such that*

1. *Σ_1 and Σ_2 have the same null space.*
2. *$\Sigma_2 = \Sigma_1 + ww^T$ for some $w \in \mathbb{R}^d$.*

Then we have

$$d_{TV}\left(\mathcal{N}(\mu, \Sigma_1), \mathcal{N}(\mu, \Sigma_2)\right) = \Theta\left(\min\left\{1, w^T \Sigma_1^\dagger w\right\}\right).$$

Theorem 2.3.11 (Chernoff bound for univariate Gaussians, Theorem 9.3 of [MU17])**.** *Let X be a univariate Gaussian with mean μ and variance $\sigma^2 > 0$: $X \sim \mathcal{N}(\mu, \sigma^2)$. Then for any $a > 0$*

$$\Pr\left[|X - \mu| \geqslant a\sigma\right] \leqslant 2e^{-a^2/2}. \tag{2.4}$$

2.3.3. ℓ_2-heavy hitter, ℓ_p-sampler and ℓ_2 estimation

Proposition 2.3.12 (ℓ_2-heavy hitters [KLM$^+$14])**.** *For any $\eta > 0$, there is a decoding algorithm D and a distribution on matrices $A \in \mathbb{R}^{O(\eta^{-2}\operatorname{polylog}(N)) \times N}$ such that, for any $x \in \mathbb{R}^N$, given Ax, the algorithm D returns a vector $\tilde{x}$ such that $\tilde{x}$ has $O(\eta^{-2}\operatorname{polylog}(N))$ non-zeros and satisfies*

$$\|x - \tilde{x}\|_\infty \leqslant \eta \|x\|_2$$

with high probability over the choice of A. The sketch Ax can be maintained and decoded in $O(\eta^{-2}\operatorname{polylog}(N))$ space.

Given a vector x of size N and a number $\delta > 0$, an ℓ_p sampler is an algorithm that output an index i with probability

$$p_i \in (1 \pm N^{-c})\frac{|x_i|^p}{\|x\|_p^p} \pm O(N^{-c})$$

where c is arbitrary constant. The algorithm may also output Fail with probability at most δ.

For any $0 \leq p \leq 2$, there is a polylog size linear sketch for ℓ_p-sampler.

Proposition 2.3.13 ([JST11, CF14]). *For any constant c and $0 < \delta < 1$, there is a linear sketch for ℓ_0-sampling with measurement size $O(\log^2 n \log 1/\delta)$.*

Proposition 2.3.14 ([JW21]). *For any $0 < p < 2$, any $\varepsilon, \delta_1, \delta_2 > 0$ and any constant $c > 0$, there is a linear sketch of size $O(\log^2 N(\log\log N)^2 \log(1/\delta_1) + \varepsilon^{-p} \log N \log^2(1/\delta_2) \log(1/\delta_1))$ such that given an N-dimensional vector x, with probability $(1 - \delta_1)$, it can recover an index i, such that the probablity of outputting i is*

$$p_i \in (1 \pm N^{-c}) \frac{|x_i|^p}{\|x\|_p^p} \pm O(N^{-c})$$

Moreover, if the sketch does output an index i, then it also recovers a value x_i' such that $|x_i| \leq x_i' \leq (1 + \varepsilon)|x_i|$, with probability $1 - \delta_2$.

Proposition 2.3.15 ([KLM+14]). *For any $\varepsilon > 0$ and any constant $c > 0$, there is a linear sketch of size $O(\varepsilon^{-2} \operatorname{polylog}(N))$ such that given an N-dimensional vecotr x, with probability N^{-c}, we can recover a vector x' such that $\|x - x'\|_\infty \leq \varepsilon \|x\|_2$.*

Proposition 2.3.16 ([JL84]). *For any $0 < \delta < 1$ and any constant $c > 0$, there is a linear sketch of size $O(\delta^{-2} \log N)$ such that given an N-dimensional vector x, with probability N^{-c}, we can recover a vector x' such that $\|x'\|_2 \in (1 \pm \delta)\|x\|_2$.*

2.4. Graph Sparsification

2.4.1. Edge strengths and cut sparsifiers

Given a graph G, a *k-strongly connected component* is a maximal vertex induced subgraph whose minimum cut size is at least k. Thus, all k-strongly connected components form a partition of the entire vertex set. The following fact gives an equivalent way of obtaining the k-strongly connected components.

Fact 2.4.1. *Given any graph and integer k, consider a process where we iteratively remove (the edges across) an arbitrary cut of size strictly smaller than k, until there is no such cut left. Then the connected components in the resulting graph are the k-strongly connected components of the*

original graph.

The *strength* of an edge e in the graph, denoted k_e, is the maximum value of k such that a k-strong component of G contains both endpoints of e. The weighted sum of the inverse of the strength of every edge in a graph is at most $n-1$.

Claim 2.4.2 ([BK15])**.** *For any weighted graph $G = (V, F, w)$ on n vertices, $\sum_{f \in F} \frac{w_f}{k_f} \leqslant n-1$.*

Given a graph G, an $(1 \pm \varepsilon)$-cut sparsifier G' is a subgraph of G such that any cut in G is preserved in G' to within a factor of $(1 \pm \varepsilon)$. The seminal result [BK15] shows that, for any graph G, if we construct a graph G' as follows: we include each edge e in G' with probability $p_e \geq \Omega(\frac{w_e \log n}{\varepsilon^2 k_e})$, and give weight $\frac{w_e}{p_e}$ if it gets chosen, then G' is a $(1 \pm \varepsilon)$ cut sparsifier of G with high probability.

2.4.2. Graph matrices, leverage scores, and spectral sparsifiers

Fix an arbitrary orientation of all possible $\binom{n}{2}$ edge slots of the graph. Let $B \in \mathbb{R}^{\binom{n}{2} \times n}$ be the *edge-vertex incidence matrix* of an undirected, unweighted complete graph over n vertices. That is, for every edge $e = (u, v)$ oriented from $u \to v$, there is a row $b_e \in \mathbb{R}^n$ in B corresponding to e such that the column u has value 1, the column v has value -1, and all other columns have value 0. We also write b_e as $b_{u,v}$. For a graph G, we write $B_G \in \mathbb{R}^{\binom{n}{2} \times n}$ to denote the matrix obtained from B by zeroing out rows corresponding to absent edges in G.

Given any weighted graph G, let $W_G \in \mathbb{R}^{\binom{n}{2} \times \binom{n}{2}}$ be the diagonal matrix whose diagonal entries are the weights of the edges corresponding to them, i.e. $(W_G)_{ee} = w_e$. If an edge e is not present in the graph, then $(W_G)_{ee} = 0$. The *Laplacian matrix* of G is given by $L_G = B_G^T W_G B_G = (W_G^{1/2} B_G)^T (W_G^{1/2} B_G)$. Notice that for unweighted graphs, we have $L_G = B_G^T B_G$.

For any edge e, its *leverage score* τ_e is given by $\tau_e = (\sqrt{w_e} b_e)^T L_G^\dagger (\sqrt{w_e} b_e) = w_e b_e^T L_G^\dagger b_e$, where $L_G^\dagger$ is the Moore-Penrose pseudoinverse of the Laplacian matrix L_G.

Fact 2.4.3. *The sum of the leverage scores of all edges $\sum_e \tau_e = \operatorname{rank}(W_G^{1/2} B_G) \leq n-1$.*

In general, for *any* matrix $C \in \mathbb{R}^{m \times n}$ whose i^{th} row is denoted by $c_i \in \mathbb{R}^n$, we define the leverage score of the i^{th} row by $\tau_i = c_i^T (C^T C)^\dagger c_i$, and we once again have $\sum_{i=1}^m \tau_i = \operatorname{rank}(C)$.

If we view the graph G as an electrical network where each edge e has resistance $1/w_e$, then the *effective resistance* between the two vertices s, t is given by $r_{s,t} = b_{s,t}^T L_G^\dagger b_{s,t}$. For an edge $e = (s, t)$,

we also define its effective resistance as $r_e = r_{s,t}$. Thus, the leverage score $\tau_e = w_e r_e$.

If we inject f units of electrical flow into a vertex s and extract 1 unit from a vertex t, then $fL^\dagger b_{s,t} \in \mathbb{R}^n$ is referred to as *the set of vertex potentials induced by the electrical flow*. The relation between vertex potentials of the electrical flow is characterized by Ohm's Law.

Fact 2.4.4 (Ohm's Law)**.** *Let $x = fL^\dagger b_{s,t} \in \mathbb{R}^n$ be the set of vertex potentials when we send f units of electrical flow from s to t. Then we have $x_s - x_t = f(b_{s,t} L^\dagger b_{s,t})$. Moreover, for any edge $e = (u, v)$ with $x_u \geqslant x_v$, the flow on this edge is in the direction of $u \to v$ and has amount exactly $w_e(x_u - x_v)$.*

It is also known that the vertex potentials induced by an electrical flow minimizes the total energy. Specifically, for an arbitrary set of vertex potentials $x \in \mathbb{R}^n$, we define its *energy* to be $x^T L_G x = \sum_{e=(u,v)\in G} w_e(x_u - x_v)^2$, and define its *normalized energy with respect to vertices s, t* to be $\frac{x^T L_G x}{(x^T b_{s,t})^2}$ (i.e. the energy divided by $(x_s - x_t)^2$). Then we have:

Fact 2.4.5. *The vertex potentials induced by an electrical flow from s to t minimizes the normalized energy with respect to s, t. That is, for any $f > 0$, we have*

$$fL^\dagger b_{s,t} \in \arg\min_{x\in\mathbb{R}^n} \frac{x^T L_G x}{(x^T b_{s,t})^2},$$

and thus, by plugging in $x = L^\dagger b_{s,t}$, the smallest normalized energy w.r.t. s, t is

$$\min_{x\in\mathbb{R}^n} \frac{x^T L_G x}{(x^T b_{s,t})^2} = \frac{b_{s,t} L_G^\dagger L_G L_G^\dagger b_{s,t}}{(b_{s,t}^T L_G^\dagger b_{s,t})^2} = \frac{b_{s,t} L_G^\dagger b_{s,t}}{(b_{s,t}^T L_G^\dagger b_{s,t})^2} = \frac{1}{b_{s,t}^T L_G^\dagger b_{s,t}},$$

exactly 1 *over the effective resistance between s, t.*

Given a graph G with Laplacian matrix L_G, a $(1\pm\varepsilon)$-spectral sparsifier G' is a graph with Laplacian matrix $L_{G'}$ such that for any $x \in \mathbb{R}^n$, $x^T L'_G x \in (1 \pm \varepsilon) x^T L_G x$. In other words, $\frac{1}{1+\varepsilon} L_G \preceq L_{G'} \preceq (1 + \varepsilon) L_G$ (recall Definition 2.3.2). We also use $L_G \approx_{1+\varepsilon} L_{G'}$ to denote the same relation between L_G and $L_{G'}$. For two scalars $a, b \geqslant 0$, we also write $a \approx_{1+\varepsilon} b$ to denote $\frac{a}{1+\varepsilon} \leqslant b \leqslant (1 + \varepsilon)a$.

If we sample each edge of a graph with probability proportional to its leverage score or larger, and reweight it accordingly, then with high probability we get a spectral sparsifier. In fact, this sampling

process gives a good spectral approximation for any $C \in \mathbb{R}^{m\times n}$

Theorem 2.4.6 ([SS11a, Tro12]). *Let $\varepsilon > 0$. Given a matrix $C \in \mathbb{R}^{m\times n}$, let $p_1, \ldots, p_m \in \mathbb{R}$ be such that $1 \geqslant p_i \geqslant \min\left\{1, 100\tau_i\varepsilon^{-2}\log n\right\}$ for all $i \in [m]$. Let $\tilde{W} \in \mathbb{R}^{m\times m}$ be a diagonal matrix such that $\tilde{W}_{ii} = 1/p_i$ with probability p_i and $\tilde{W}_{ii} = 0$ otherwise. Then with high probability,*

$$C^T\tilde{W}C \approx_\varepsilon C^TC.$$

Recall that when $C = W_G^{1/2}B_G$, we have $L_G = C^TC$. Thus if we sample each edge e with probability $p_e = \min\left\{1, 100\tau_e\varepsilon^{-2}\log n\right\}$ and reweight it to w_e/p_e if sampled, we get, by the claim above and that $\sum_e \tau_e \leqslant n-1$, a $(1+\varepsilon)$-spectral sparsifier of $O(n\varepsilon^{-2}\log n)$ edges. We could also do an oversampling, where we sample each edge e with some probability $p_e \geqslant \min\left\{1, 100\tau_e\varepsilon^{-2}\log n\right\}$ and also reweight it to w_e/p_e if sampled, and then we get a $(1+\varepsilon)$-spectral sparsifier of $O(\sum_e p_e)$ edges.

2.5. Capacitated Graphs and Maximum Flows

Graphs. We represent an n-vertex, m-edge undirected, capacitated graph as $G = (V, E, c)$ with V being the vertices (nodes), E being the edges, and c being the edge capacities (weights). We also use $V(G)$ and $E(G)$ to denote the vertices and edges of the graph G, respectively. We will interchangeably write the capacity of an edge e as $c(e)$ or c_e. Sometimes the same edge e appears in two different graphs with different capacities, thus we write $c^H(e)$ or c_e^H to denote the weight of e in graph H to avoid ambiguity. In Section C.5, we show how to transform a graph with arbitrary capacities into one with polynomial aspect ratio while only reducing the maximum s-t flow by a $(1-\varepsilon)$ factor, for any $s, t \in G$ and $\varepsilon > 0$. Thus, throughout the paper we assume that graphs have $\mathrm{poly}(n/\varepsilon)$ aspect ratio.

We usually need to solve maximum flows by adding a source and a sink, for which we set up the following notation.

Definition 2.5.1 (Graphs with Sources and Sinks). Given a graph H with two (not necessarily disjoint) vertex subsets $V_1, V_2 \subseteq V(H)$, and a parameter $c_u \geqslant 0$ for each $u \in V_1$, and a parameter $d_u \geqslant 0$ for each $u \in V_2$, we write H_{st} to denote the graph obtained from H by doing the following:

1. Add a vertex s, and connect s to each $u \in V_1$ with capacity c_u;
2. Add a vertex t, and connect t to each $u \in V_2$ with capacity d_u.

Congestion Approximators and Sherman's Algorithm. Given a graph $G = (V, E, c)$ and a demand vector b, let $\mathsf{opt}_G(b) = \min_{f:Bf=b} \|C^{-1}f\|_\infty$ be the optimal congestion of any flow routing a given demand vector b in the graph G, with B being the edge-vertex incidence matrix of G.

We first define congestion approximators. Note that in this paper, we *always* consider congestion approximators that are trees, which are readily plugged into Sherman's algorithm.

Definition 2.5.2 (Congestion Approximators)**.** Given a graph $G = (V, E, c)$, a tree $R = (V_R, E_R, c_R)$ with $V \subseteq V_R$ is an α-congestion approximator of G for some $\alpha \geqslant 1$ if for any demand vector b, we have

$$\mathsf{opt}_R(b') \leqslant \mathsf{opt}_G(b) \leqslant \alpha \cdot \mathsf{opt}_R(b'),$$

where b' is obtained by appending b with zeros on entries $V_R \setminus V$.

We next state the parallel version of Sherman's algorithm. While we believe it is known that Sherman's algorithm can be implemented with low depth *modulo the computation of a good congestion approximator*, we give a discussion of such an implementation in Appendix C.3 for completeness.

Theorem 2.5.3 (Sherman's algorithm [She13b], Parallel Version)**.** *There is a PRAM algorithm that, given a graph $G = (V, E, c)$, an α-congestion approximator $R = (V_R, E_R, c_R)$ of G, a source $s \in V$ and a sink $t \in V$, and an $\varepsilon \in (0, 1)$, computes a $(1-\varepsilon)$-approximate maximum s-t flow as well as a $(1+\varepsilon)$-approximate minimum s-t cut in G. The total work of the algorithm is $O((|E| + |E_R|)\alpha^2\varepsilon^{-3}\operatorname{polylog}(n))$, and the depth of the algorithm is $O(\alpha^2\varepsilon^{-3}\operatorname{polylog}(n))$.*

Contracted Subgraphs. We give the definition of contracted subgraphs below. We use X to denote the subset of vertices that we contract, and use $S = V(G) \setminus X$ to denote the remaining vertices.

Definition 2.5.4 (Contracted Subgraphs)**.** Given a graph $G = (V, E, c)$ and a subset of vertices $S \subseteq V(G)$, we denote by $G(S)$ the graph obtained from G by contracting $X := V \setminus S$ into a single

supernode u_X, deleting any resulting self-loops, and keeping all resulting parallel edges.

We now define reweighted contracted subgraphs which will be useful in our new congestion approximator construction.

Definition 2.5.5 (Reweighted Contracted Subgraphs)**.** Given a graph $G = (V, E, c)$ and a pair $\mathcal{P} = (S, \omega)$, where $S \subseteq V(G)$ is a subset of vertices, and $\omega : E(S, V(G) \setminus S) \to (0, 1]$ is a real-valued *reweighting function*, we denote by $G(\mathcal{P})$ the graph obtained from G by doing the following operations:

1. Contract $X := V \setminus S$ into a single supernode u_X, deleting any resulting self-loops, and keeping all resulting parallel edges;
2. Scale the weight of each (parallel) edge e incident on u_X by a factor of $\omega(e)$.

We call u_X the *contracted vertex*, and refer to other vertices (a.k.a. S) in $G(\mathcal{P})$ as *uncontracted vertices*. We refer to the edges in $G(\mathcal{P})$ that are incident on u_X the *boundary edges*.

Subdivision Graphs and Well-Linked Edges. We recall the definition of subdivision graphs from [RST14].

Definition 2.5.6 (Subdivision Graphs)**.** Given a capacitated graph $G = (V, E, c)$, we write $G' = (V', E', c')$ to denote the subdivision graph of G, defined as follows. To obtain G', we split each edge $e = (u, v)$ in G by introducing a new *split vertex* (or *subdivision vertex*) x_e and two *split edges* (u, x_e) and (x_e, v), both with capacity c_e. In words, we have $V' = V \cup X_E$ where X_E is the set of split vertices of the edges in E, $E' = \bigcup_{e=(u,v)\in E} \{(u, x_e), (x_e, v)\}$, and $c'_{(u,x_e)} = c'_{(v,x_e)} = c_e$ for every edge $e \in E$.

For a subset of edges $F \subseteq E$, we write X_F to denote the set of split vertices of edges in E, and c_F (or equivalently, $c(F)$) to denote the total capacity of edges in F.

Definition 2.5.7. Given $G(\mathcal{P})$ and another reweighting function $\tilde{\omega}$ supported on the split edges of the boundary edges of $G(\mathcal{P})$ in $G'(\mathcal{P})$. Then we define the graph $G'(\tilde{\mathcal{P}} = (\mathcal{P}, \tilde{\omega}))$ to be the graph obtained from $G'(\mathcal{P})$ by reweighting the split edges of the boundary edges by $\tilde{\omega}$. That is, for an edge e that is a split edge of a boundary edge, we have $c_e^{G'(\tilde{\mathcal{P}}=(\mathcal{P},\tilde{\omega}))} = \tilde{\omega}(e) \cdot c_e^{G'(\mathcal{P})}$.

We next recall the notion of well-linked edges as used in [RST14].

Definition 2.5.8 (Well-linked Edges)**.** For a capacitated graph $G = (V, E, c)$, a set $F \subseteq E$ of edges of total capacity c_F is called *β-well-linked* for some $\beta \in (0, 1]$ if there is a feasible multicommodity flow in the subdivision graph G' that sends $(\beta c_e c_f / c_F)$ units of flow from x_e to x_f for all pairs $e, f \in F$ simultaneously.

Any subset of well-linked edges are also well-linked.

Proposition 2.5.9. *If set $F \subseteq E$ of edges is β-well-linked in $G = (V, E, c)$, then any subset $F' \subseteq F$ of F is β-well-linked.*

Proof. Given a product demand on F', we can first let each edge distribute its demands uniformly to all edges in F with each edge getting flow proportional to its capacity. This can be done with congestion $1/\beta$ since edges in F are β-well-linked. Then for each demand between two vertices s, t in F', we can compose the routings which distributed the flow from s and t to F. The proposition thus follows. □

2.6. PRAM Primitives

We state a few basic PRAM primitives, whose implementation is discussed in Appendix C.1 for completeness.

Proposition 2.6.1 (Prefix Sums [Akl97])**.** *There is a PRAM algorithm that, given a list of values $a_1, \ldots, a_n$, computes all prefix sums $\sum_{i \leqslant k} a_i$'s. The algorithm has $O(n)$ total work and $O(\log n)$ depth.*

Proposition 2.6.2 (Subtree Sums)**.** *There is a PRAM algorithm that given tree $T = (V_T, E_T, w_T)$ where $w_T : V_T \to \mathbb{R}$ is a vertex weight function, a vertex $r \in V_T$, computes all subtree sums of the weight function w_T with the r being the root of tree. The algorithm has total work $O(|E_t|)$ and total depth $O(\log |E_T|)$.*

2.7. Similarity Graphs and Hierarchical Clustering

We use the notation $G = (V, E)$ to represent unweighted similarity graphs (where all present similarity information are the same), and $G = (V, E, w)$ for weighted similarity graphs (where higher weights denote higher similarity). We use lowercase letters u, v to refer to vertices (data points) in

V, and given a vertex v, we use $d_G(v)$ to refer to its degree in graph G. We use capital letters S, T to represent subsets of vertices, and given a vertex set $S \subset V$, we use $|S|$ to refer to its cardinality, $\overline{S} := V \setminus S$ to refer to its complement, and $G[S]$ to refer to the subgraph of G induced by vertex set S. Furthermore, given two disjoint vertex sets S, T, we use $w_G(S,T) := \sum_{(u,v)\in E: u\in S, v\in T} w(u,v)$ to represent the total weight of the edges in graph G with one endpoint in S and the other in T. In the case of an unweighted graph, this is equivalent to the number of edges going from S to T. For ease of notation, we use $w_G(S) := w_G(S, \overline{S})$, and when the implied graph is clear from context, w_e to refer to the weight of an edge $e \in E$ in that graph.

Given a graph $G = (V, E)$, we use $\mathcal{T}$ to refer to a hierarchical clustering (tree) of the vertex set V, and $\mathsf{cost}_G(\mathcal{T})$ to refer to the cost of this clustering in graph G. Without loss of generality, we restrict our attention to just full binary hierarchical clustering trees, since the optimal tree is binary [Das16]. Any internal node S of a hierarchical clustering tree corresponds to a binary split (S_ℓ, S_r) (the left and right children of S in $\mathcal{T}$) of the set of leaves in the subtree rooted at S. With some overload of notation, we let S represent both, the internal node of the clustering tree as well as the set of leaves $S \subseteq V$ in the subtree rooted at internal node S. Furthermore, since (the leaves in the subtree rooted at) an internal node S can correspond to an arbitrary subset of V, we use the term *split* to refer to a partition (S_ℓ, S_r) of S to disambiguate it from *cuts*, which are a partition of the entire vertex set V.

Recall that ϕ is used to denote the approximation ratio of any desired offline algorithm for hierarchical clustering. For example, if allowed unbounded computation time, we have $\phi = 1$; given polynomial time, the current best algorithm [CC17] gives $\phi = O(\sqrt{\log n})$. We assume this abstraction as any improvement in the approximation ratio here automatically implies an identical improvement in our upper bounds.

We conclude the preliminaries by presenting two useful facts from [Das16]; the first is an equivalent reformulation of the similarity based hierarchical clustering cost function defined earlier in the introduction, and the second is the cost of any hierarchical clustering in an unweighted clique.

Fact 2.7.1. *The hierarchical clustering cost of any tree $\mathcal{T}$ with each internal node S corresponding*

to a binary split (S_ℓ, S_r) of the subset $S \subseteq V$ of vertices is equivalent to the sum

$$\mathsf{cost}_G(\mathcal{T}) = \sum_{\text{splits } S \to (S_\ell, S_r) \text{ in } \mathcal{T}} |S| \cdot w_G(S_\ell, S_r)\,.$$

Fact 2.7.2. *The cost of any hierarchical clustering in an unweighted n-vertex clique is $(n^3 - n)/3$.*

CHAPTER 3

AN IMPROVED MAXIMUM MATCHING ALGORITHM IN THE STREAMING MODEL

In this chapter, we present our improved maximum matching algorithm in the streaming model. The results from this chapter are based on my work [ABKL23] joint with Assadi, Behnezhad, and Khanna.

We start by giving a technical overview of the main ideas of our algorithm in Section 3.1. Then in Section 3.2 we introduce a new construction of a matching sparsifier that we call *matching cover*, which is a sparse subgraph that approximately preserves the size of every matching in the original graph. Finally, leveraging this new matching cover construction, we present our improved streaming algorithm for the maximum matching problem in Section 3.3.

3.1. Overview of the Algorithm

The key behind our improved maximum matching algorithm in the streaming model is a new construction of a matching sparsifier. Therefore, we start by introducing the notion of such sparsifiers.

Matching sparsifiers, which loosely speaking, are sparse subgraphs that approximately preserve the maximum matching have long been known to be an important tool for (variants of) streaming algorithms. Some prominent examples include edge-degree constrained subgraphs (EDCS) [BS15, BS16] and its generalizations [AB19, BK22], kernels [BHN16, BHI18, ACC$^+$18, BDL21], and matching skeletons [GKK12]. One of our main contributions, and the key to Result 1, is a new matching sparsifier based on Szemerédi's Regularity Lemma.

Our matching sparsifier, more strongly, is a *matching cover*—à la [GKK12]—which not only preserves an approximate *maximum* matching of the graph, but rather "covers" smaller matchings of it as well. Let us formalize this. For a given graph G, we write $\mu(G)$ to denote the maximum matching size of G, and write $G[A, B]$ to denote the bipartite subgraph of G between some disjoint vertex subsets A, B. We say a subgraph H of G is an **α-matching cover** for $\alpha \in (0, 1)$ if for any disjoint subsets of vertices (A, B) in G, $\mu(H[A, B]) \geqslant \mu(G[A, B]) - \alpha \cdot n$. That is, H preserves the largest matching in

the induced bipartite subgraph $G[A, B]$ to within an additive $\alpha \cdot n$ factor. While from an information theoretic perspective, *existence* of an $o(n^2)$-edge $o(1)$-matching cover for bipartite graphs was proved in the original paper of [GKK12], it was not known up until now whether one can find such matching covers efficiently, say in polynomial time.

In this paper, we prove that there is an $\tilde{O}(n^\omega)$-time[5] offline algorithm that computes an $o(n^2)$-edge $o(1)$-matching cover of any n-vertex graph (not necessarily bipartite). Our algorithm builds on Szemerédi's Regularity Lemma (and its algorithmic version due to [ADL$^+$92]). We first explain how our offline algorithm for obtaining a matching cover works, and then outline its use in obtaining improved streaming algorithms.

3.1.1. Matching Covers via Regularity Lemma

Roughly speaking, Szemerédi's regularity lemma [Sze75] says that the vertices of any graph can be partitioned into a small *irregular* part C_0 with $|C_0| = o(n)$, plus k other equal-size parts $C_1, \ldots, C_k$ for some $k \in [\omega(1), \log n]$. The latter k parts have the property that all but $o(1)$-fraction of the C_i, C_j pairs are *regular*: for any pair of subsets $X \subseteq C_i, Y \subseteq C_j$ with large enough size, the edge density between X, Y is similar to that of C_i, C_j. Therefore, if the edges between C_i, C_j are dense to start with, the density will also be high between every large enough $X \subseteq C_i, Y \subseteq C_j$ pair.

It is not difficult to see that by regularity, any large matching between a dense regular pair C_i, C_j can be mostly preserved if we subsample edges between them at a sufficiently high rate $p = o(1)$. In particular, the subsampled graph will be a matching cover of the graph induced by edges between the regular pair C_i, C_j. This suggests a natural strategy for building an $o(1)$-matching cover with $o(n^2)$ edges: *subsample* the edges between dense regular C_i, C_j pairs at rate $p = o(1)$ and take *all* other edges. We would like to show that this is an α-matching cover for some $\alpha = o(1)$.

This idea runs into the following problem. Suppose we have an (αn)-size matching M whose edges are evenly distributed across all $\binom{k}{2}$ pairs of C_i, C_j, then the number of edges of M between each C_i, C_j pair is only $O(\alpha \cdot n/k^2)$. This means that only an $O(\alpha/k)$ fraction of vertices in C_i, C_j are matched to each other – this is unfortunately way too small to invoke the regularity property.

[5]Here and throughout, $\omega \approx 2.37286$ is the matrix multiplication exponent with current best bounds achieved by [AW21].

We get around this issue by first focusing on solving an *α-hitting set* problem: find one edge between endpoints of any (αn)-size matching – we will show later on using a similar argument as in [GKK12] that this is sufficient for obtaining an α-matching cover. Now to fix our problem about an (αn)-size matching whose edges are distributed across many pairs, we present a strategy for *consolidating the support* of a matching over different pairs. This consolidation argument shows that whenever there is a large matching M between dense regular C_i, C_j pairs, there must also exist another (almost as) large matching M' that is supported on the same set of vertices $V(M)$ but only uses edges between a small number of such C_i, C_j pairs. As a result, there must exist one pair of C_i, C_j where a substantial fraction of vertices are matched to each other, to which we are now able to apply regularity to prove the existence of an edge between them in the subsampled graph (which solves our α-hitting set problem). At a high level, our argument for consolidating the support of the matching is proved by (i) viewing the matching M as a *fractional* matching in a meta graph obtained by contracting each C_i into a supernode; and (ii) rounding the fractional matching by an edge sampling process.

All in all, using the algorithm of [ADL$^+$92] for finding the regularity lemma partition in $\widetilde{O}(n^\omega)$ time, and a direct sampling algorithm between dense regular pairs, this step gives us an $\widetilde{O}(n^\omega)$ time and $\widetilde{O}(n)$ space algorithm for finding an α-matching cover of size $n^2/(\log^* n)^{\Omega(1)}$ for some $\alpha = 1/(\log^* n)^{\Omega(1)}$.

3.1.2. Applications of Matching Cover to the Streaming Setting

Our streaming algorithm in Result 1 is based on using matching covers as a natural sparsifier for matchings. The algorithm works through a series of buffers of edges $B_1, B_2, \ldots,$. The first buffer B_1 reads the edges from the input until it gets "full", i.e., receives some $o(n^2)$ edges (which is some constant factor larger than the size of our matching cover). At that point we compute a matching cover of the edges in the buffer using an offline/non-streaming algorithm and send its edges to the buffer B_2; then, we "restart" B_1 by emptying all its current edges and letting it collect more edges from the stream. The same approach is repeated across all other buffers as well. The number of these buffers can be bounded as only a constant fraction of edges in one buffer can make their way to the next one, eventually reaching a buffer that never gets full. This also implies that fewer edges will be be further "sparsified" in each matching cover, thus the error occurred due to the

approximation guarantee of the matching cover does not get amplified "too much". Thus, using this algorithm along with our matching cover algorithm for regularity lemma, leads to an $o(n^2)$-space $(1-o(1))$-approximation algorithm for single-pass streaming matchings.

One key step in making the above algorithms work is to store the $o(n^2)$ edges they have in the buffers more efficiently than spending $\Theta(\log n)$ bits per each (which is prohibitive for us given the extremely small improvement in the space the algorithms get over the trivial $O(n^2)$ bound). This is done by storing the edges via the succinct dynamic dictionary of [RR03] (see Section 2.2.3) and then performing all the computation in this compressed space instead.

3.2. A Matching Cover via Regularity Lemma

In this section, we give a polynomial time algorithm for constructing an *matching cover* of size $o(n^2)$. We will use the algorithm of this section in the streaming model.

Let us start by formally defining matching covers.

Definition 3.2.1 ([GKK12])**.** A subgraph H of an n-vertex graph G is an α-matching cover of G if for any disjoint subsets of vertices (A, B) in G, we have $\mu(H[A,B]) \geqslant \mu(G[A,B]) - \alpha \cdot n$.

The following theorem is our main result of this section.

Theorem 3.2.2. *Given any n-vertex graph G, for some $\alpha = (\log^* n)^{-\Omega(1)}$, there is an $O(n^\omega \log n)$ time algorithm, formalized below as Algorithm 3.2.1, for finding an α-matching cover H of G with at most $n^2/(\log^* n)^{\Omega(1)}$ edges.*

Even though existence of $o(n^2)$ size $o(1)$-matching covers for bipartite graphs was already proved by [GKK12], it was not known whether it is possible to find one in polynomial time (nor whether they also exist for general, not necessarily bipartite, graphs).

3.2.1. First Step: A Hitting Set Argument

In this section, we give an algorithm for finding an *α-hitting set*, defined below. We later show in Section 3.2.2 that this can be turned into a matching cover.

Definition 3.2.3. A subgraph H of an n-vertex graph G is an α-hitting set of G if for any disjoint subsets of vertices (A, B) in G satisfying $|A| = |B| = \alpha n$ and $\mu(G[A,B]) = \alpha n$, there is at least one

edge between A and B in H.

The following lemma is our main guarantee of this section.

Lemma 3.2.4. *Given any n-vertex graph G, for some $\alpha = (\log^* n)^{-\Omega(1)}$, there is an $O(n^\omega \log n)$ time algorithm, formalized below as Algorithm 3.2.1, for finding an α-hitting set H of G with at most $n^2/(\log^* n)^{\Omega(1)}$ edges.*

Proposition 2.2.3 below formalizes our algorithm for Lemma 3.2.4.

Algorithm 3.2.1. The construction of the matching cover for Theorem 3.2.2.

Let $t \leftarrow (\log^* n)^\delta$, $\gamma \leftarrow (\log^* n)^{-\delta}$ for some constant $\delta \in (0,1)$ such that $Q(t, 1/\gamma) \leqslant \log n$.

(i) Run the algorithm in Proposition 2.2.3 to obtain a γ-regular partition $C_0, \ldots, C_k$ with $t = (\log^* n)^\delta \leqslant k \leqslant Q(t, 1/\gamma) \leqslant \log n$.

(ii) Let (C_i, C_j) for $i \neq j$. We say (C_i, C_j) is *good* if (i) $i, j \neq 0$, (ii) it is γ-regular, and (iii) $d(C_i, C_j) \geqslant 8\gamma$. Otherwise, we say (C_i, C_j) is *bad*.

(iii) Let $F \subseteq E$ be a subset of edges obtained by $F := F_1 \cup F_2 \cup F_3$ where

  (a) F_1 contains the edges between the bad pairs; i.e. $F_1 := \bigcup_{\text{bad } C_i, C_j} E \cap (C_i \times C_j)$.

  (b) F_2 contains the edges within each class; i.e. $F_2 := \bigcup_{0 \leqslant i \leqslant k} E \cap (C_i \times C_i)$.

  (c) F_3 is obtained by sampling the edges between good pairs with probability $p := \frac{10}{\log n}$; i.e. $F_3 = \bigcup_{\text{good } C_i, C_j} (E \cap (C_i \times C_j))[p]$.

(iv) Return F.

It is not hard to see that Algorithm 3.2.1 outputs a subgraph with $O(\gamma n^2) = o(n^2)$ edges, since essentially the dense parts of the decomposition are subsampled and there are 'few' other edges in the graph. The following claim formalizes this.

Claim 3.2.5. *The output F of Algorithm 3.2.1, w.h.p., has at most $O(\gamma n^2)$ edges.*

Proof. Let us first count the number of edges in F_1 and F_2. We do so by counting the number of different types of bad edges separately.

1. **Edges within each class:** The total number is bounded by

$$\binom{|C_0|}{2} + \sum_{i=1}^{k} \binom{|C_i|}{2} \leqslant \binom{\gamma n}{2} + k\binom{n/k}{2} < \gamma^2 n^2 + n^2/k \leqslant \gamma^2 n^2 + n^2 (\log^* n)^{-\delta}.$$

2. **Edges between C_0 and other C_i's:** The total number is bounded by

$$\sum_{i=1}^{k} |C_0||C_i| \leqslant |C_0| n \leqslant \gamma n^2.$$

3. **Edges between irregular or sparse (i.e. density $< 8\gamma$) C_i, C_j's:** The total number is bounded by

$$\gamma \binom{k}{2} (n/k)^2 + \sum_{1 \leqslant i < j \leqslant k : d(C_i, C_j) < 8\gamma} |E(C_i, C_j)| \leqslant 5\gamma n^2.$$

Summing up these and noting that $\gamma = (\log^* n)^{\delta}$ implies that $|F_1| + |F_2| \leqslant O(\gamma n^2)$.

Moreover, since F_3 includes each good edge independently with probability $p = 10/\log n$, a simple Chernoff bound implies that w.h.p. $|F_3| \leqslant O(n^2/\log n) \ll \gamma n^2$, completing the proof. □

The harder part of the proof, is to show that the sparse subgraph returned by Algorithm 3.2.1 is indeed a matching cover. We continue with the following claim.

Claim 3.2.6. *W.h.p., it holds for all $X \subseteq C_i, Y \subseteq C_j$ such that (C_i, C_j) a good pair, $|X| \geqslant \gamma |C_i|$, and $|Y| \geqslant \gamma |C_j|$ that $|F_3(X, Y)| > n^2/\log^6 n$.*

Proof. Take a good pair (C_i, C_j) and subsets $X \subseteq C_i, Y \subseteq C_j$ satisfying $|X| \geqslant \gamma |C_i|, |Y| \geqslant \gamma |C_j|$. Since C_i, C_j is good, we know that $d(C_i, C_j) \geqslant 8\gamma$ and so $d(X, Y) \geqslant d(C_i, C_j) - \gamma \geqslant 7\gamma$ by the guarantee of the regularity lemma for good pairs. Thus, the number of edges between X, Y in E is

$$|E(X, Y)| \geqslant d(X, Y)|X||Y| \geqslant 7\gamma(\gamma|X|)(\gamma|Y|) \geqslant 7\gamma^3 (n/k)^2 \geqslant 7(\log^* n)^{-3\delta} (n/\log n)^2 \geqslant 7n^2/\log^5 n.$$

Since in obtaining F we sample edges between C_i, C_j with probability $p = 10/\log n$, we get that

$$\mathbf{E}|F(X, Y)| = p|E(X, Y)| \geqslant 7n^2/\log^6 n.$$

Defining $\delta := \sqrt{6n/\mathbf{E}|F(X,Y)|}$ and applying the Chernoff bound, we get that

$$\Pr[|F(X,Y)| < (1-\delta)\mathbf{E}|F(X,Y)|] \leqslant \exp\left(-\frac{\delta^2 \mathbf{E}|F(X,Y)|}{2}\right) < e^{-3n}.$$

Using the lower bound $7n^2/\log^6 n$ for $\mathbf{E}|F(X,Y)|$ and noting that δ is much smaller than, say, $1/2$ precisely because of this lower bound, we get that

$$\Pr\left[|F(X,Y)| < \frac{n^2}{\log^6 n}\right] < e^{-3n}.$$

Since the total choices of C_i, C_j and subsets X, Y is less than $\log^2 n \times 2^n \times 2^n < e^{2n}$, by a union bound we get that $|F(X,Y)| \geqslant n^2/\log^6 n$ for all X and Y and all C_i and C_j with probability at least $1 - \frac{e^{2n}}{e^{3n}} \geqslant 1 - 2^{-n}$. □

Remark 3.2.7. *For the purpose of this section, we will only use $|F_3(X,Y)| > 0$ instead of the much larger lower bound of Claim 3.2.6 for this set. We note that for the weaker guarantee of $|F_3(X,Y)| > 0$, it suffices to only sample $\widetilde{O}(n)$ edges in Algorithm 3.2.1 instead of $O(n^2/\log n)$. Nonetheless, reducing the size of F_3 will not result in a sparser matching cover because F_1 and F_2 will already be in the order of $n^2/(\log^* n)^{\Theta(1)}$.*

Next, we prove the following lemma on *consolidating the support* of an arbitrary fractional matching so that each non-zero variable takes a sufficiently large value.

Lemma 3.2.8. *Let $\mathbf{x}$ be any fractional matching (not necessarily in the matching polytope). For any $\varepsilon \in (0,1]$, there is a fractional matching $\mathbf{y}$ such that all the following hold:*

1. *For any vertex v, $y_v \leqslant x_v$, where here $y_v := \sum_{e \ni v} y_e$ and $x_v := \sum_{e \ni v} x_e$.*
2. *$\text{supp}(\mathbf{y}) \subseteq \text{supp}(\mathbf{x})$. That is, if $y_e > 0$ for some edge e, then $x_e > 0$.*
3. *For any edge e, either $y_e = 0$ or $y_e \geqslant \frac{\varepsilon^3}{12\ln(1/\varepsilon)}$.*
4. *$|\mathbf{y}| \geqslant |\mathbf{x}| - 2\varepsilon n$, where here $|\mathbf{y}| := \sum_e y_e$ and $|\mathbf{x}| := \sum_e x_e$.*

Proof. Let $\beta = 6\ln(1/\varepsilon)/\varepsilon^3$; assume for ease of exposition that β is an integer. For every edge $e \in E$ and every $i \in [\beta]$, draw a Bernoulli random variable b_e^i that is 1 with probability x_e. Now

for any edge e we define $z_e := \beta^{-1} \sum_{i=1}^{\beta} b_e^i$, that is, z_e is the fraction of the β trials $b_e^1, \ldots, b_e^\beta$ that succeed. Moreover, for every vertex v we let $z_v := \sum_{e \ni v} z_e$, and construct $\mathbf{y}$ as follows:

$$y_e = \mathbb{1}[z_v < (1+\varepsilon)x_v \text{ and } z_u < (1+\varepsilon)x_u] \cdot z_e/(1+\varepsilon) \qquad \text{for any edge } e. \tag{3.1}$$

First, observe that if $z_v \geqslant (1+\varepsilon)x_v$ for a vertex v, then $y_v = 0$ by construction. Otherwise, we have $y_v \leqslant z_v/(1+\varepsilon) < (1+\varepsilon)x_v/(1+\varepsilon) = x_v$. This proves the first property and also the fact that $\mathbf{y}$ is a fractional matching.

Second, note that if $x_e = 0$, then $z_e = 0$ and so $y_e = 0$. This immediately implies that any edge in the support of $\mathbf{y}$ must also belong to the support of $\mathbf{x}$, proving the second property.

Third, observe from the definition of z_e that either $z_e = 0$ or $z_e \geqslant \beta^{-1} = \varepsilon^3/6\ln(1/\varepsilon)$. This implies that either $y_e = 0$ or $y_e \geqslant \varepsilon^3/6\ln(1/\varepsilon)(1+\varepsilon) \geqslant \varepsilon^3/12\ln(1/\varepsilon)$, proving the third property.

We now turn to the fourth property. Take a vertex v. In the event that $z_v \geqslant (1+\varepsilon)x_v$, we charge all of z_v to v. That is, we define the charge $\phi(v)$ to be z_v if $z_v \geqslant (1+\varepsilon)x_v$ and $\phi(v) = 0$ otherwise. We get that

$$\mathbf{E}|\mathbf{y}| = \frac{1}{2}\sum_v \mathbf{E}[y_v] \geqslant \frac{1}{2(1+\varepsilon)}\sum_v (\mathbf{E}[z_v - \phi(v)]) > (1-\varepsilon)|\mathbf{x}| - \frac{1}{2}\sum_v \mathbf{E}[\phi(v)], \tag{3.2}$$

where the latter inequality follows from $\sum_v \mathbf{E}[z_v] = \sum_v x_v = 2|\mathbf{x}|$. Next, we show that $\mathbf{E}[\phi(v)] \leqslant 2\varepsilon$ for any vertex v. Observe that since $\mathbf{E}[\phi(v)] \leqslant \mathbf{E}[z_v] = x_v$, we are done if $x_v \leqslant 2\varepsilon$. So let us assume that $x_v > 2\varepsilon$. We have

$$\begin{aligned}
\mathbf{E}[\phi(v)] &= \Pr[z_v \geqslant (1+\varepsilon)x_v] \cdot \mathbf{E}[z_v \mid z_v \geqslant (1+\varepsilon)x_v] \\
&\leqslant \mathbf{E}[z_v] - \Pr[|z_v - x_v| < \varepsilon x_v] \cdot \mathbf{E}[z_v \mid |z_v - x_v| < \varepsilon x_v] \\
&\leqslant x_v - \Pr[|z_v - x_v| < \varepsilon x_v] \cdot (1-\varepsilon)x_v.
\end{aligned} \tag{3.3}$$

Now observe that $z_v\beta$ is a sum of $\deg(v) \cdot \beta$ independent (but not i.i.d.) Bernoulli random variables (one for each of the β trials of each edge of v) with expected value $\mathbf{E}[z_v\beta] = x_v\beta$. Hence, by the

Chernoff bound

$$\Pr[|z_v\beta - x_v\beta| \geqslant \varepsilon x_v\beta] \leqslant 2\exp\left(-\frac{\varepsilon^2 x_v\beta}{3}\right) \overset{x>2\varepsilon,\beta=6\ln(1/\varepsilon)/\varepsilon^3}{\leqslant} 2\exp\left(-4\ln(1/\varepsilon)\right) < 2\varepsilon^4.$$

Dividing through by β, this means that $\Pr[|z_v - x_v| \geqslant \varepsilon x_v] < 2\varepsilon^4$ and as a result $\Pr[|z_v - x_v| < \varepsilon x_v] \geqslant 1 - 2\varepsilon^4$. Plugging this into Eq. (3.3), we get that

$$\mathbf{E}[\phi(v)] \leqslant x_v - (1-2\varepsilon^4)(1-\varepsilon)x_v \leqslant 2\varepsilon^2 x_v + \varepsilon \leqslant 2\varepsilon x_v \leqslant 2\varepsilon.$$

Plugging $\mathbf{E}[\phi(v)] \leqslant 2\varepsilon$ back into Eq. (3.2), implies that

$$\mathbf{E}|\mathbf{y}| \geqslant (1-\varepsilon)|\mathbf{x}| - \varepsilon n \geqslant |\mathbf{x}| - 2\varepsilon n.$$

Finally, observe from our earlier discussion that the first three properties all hold with probability 1. The only place where we use the randomization of the construction of $\mathbf{y}$ is for the fourth property where we showed $\mathbf{E}|\mathbf{y}| \geqslant |\mathbf{x}| - 2\varepsilon$. This suffices for our purpose, since there must exist an outcome of $\mathbf{y}$ with size as large as its expectation, while satisfying the other properties. □

We are now ready to prove that Algorithm 3.2.1 returns a $o(1)$-matching-cover w.h.p.

Lemma 3.2.9. *The output of Algorithm 3.2.1 is, with high probability, an α-hitting set of G for $\alpha = \Theta((\gamma\log(1/\gamma))^{1/3}) = (\log^* n)^{-\Omega(1)}$.*

Proof. We show that if the event of Claim 3.2.6 holds—which was shown to hold w.h.p. in the claim—then the output of Algorithm 3.2.1 is indeed an α-hitting set.

Let A and B be some arbitrary disjoint subsets of V such that $|A| = |B| = \alpha n$ and let M be a perfect matching of size αn between A and B in G. We have to show that there is at least one edge that goes from A to B in H. Observe that if there is any edge $e \in M$ that goes across a bad pair C_i, C_j or if both endpoints of e are in the same class C_i, then this edge e will be our desired edge since Algorithm 3.2.1 will add e to F. So let us assume that every edge in M belongs to a good pair. Note that if we find a good pair C_i, C_j where at least $\gamma|C_i| = \gamma|C_j|$ edges of M go from C_i to C_j, then we can immediately apply Claim 3.2.6 to get that there is one edge connecting two

endpoints of matching M in F. Unfortunately, such a pair may not always exist. Note that since there are k classes, each of the $\Theta(k^2)$ pairs might include only $\Theta(|M|/k^2) = \Theta((\gamma \log(1/\gamma)^{1/3} n/k^2)$ edges of M. This could be much smaller than $\gamma|C_i|$ which may be of size $\Omega(\gamma n/k)$. Additionally, we have to ensure that the edge that we find goes from A to B, and that it does not connect A to A or B to B. To achieve all of this, we give a more delicate argument that uses Lemma 3.2.8.

First, let us slightly modify the matching M. Call a cluster C_i *A-majority* if

$$|C_i \cap V(M) \cap A| > .75|C_i \cap V(M)|,$$

and similarly *B-majority* if

$$|C_i \cap V(M) \cap B| > .75|C_i \cap V(M)|.$$

We construct a sub-matching M' of M by removing any edge $(u, v), u \in A, v \in B$ from it where u is in a B-majority cluster or v is in an A-majority cluster. Any cluster C_i causes removal of at most .25 fraction of its matched vertices in M, hence the total number of removed vertices from M is at most $.25|V(M)|$. Each such removed vertex may remove a unique edge in M. Thus, in total, the obtained matched M' has size at least

$$|M'| \geqslant |M| - .25|V(M)| = |M| - .5|M| = |M|/2 = \alpha n/2. \tag{3.4}$$

Now for any $i, j \in [k]$ let x_{ij} denote the fraction of the vertices of C_i that are matched to C_j in M'. Note that $x_{ij} = x_{ji}$ since the classes $C_1, \ldots, C_k$ all have equal sizes. Observe also that $\mathbf{x}$ is a fractional matching of a complete graph on k vertices: For any $i \in [k]$, $x_i := \sum_j x_{ij}$ equals the fraction of the vertices of C_i that are matched by M', and so $0 \leqslant x_i \leqslant 1$. Additionally, fractional matching $\mathbf{x}$ satisfies the following:

(X1) If $x_{ij} \neq 0$, then (C_i, C_j) is a good pair. This follows from our earlier assumption that all edges of M' belong to good pairs.

(X2) $|\mathbf{x}| \geqslant \alpha k/2$. This holds because

$$|\mathbf{x}| = \sum_{i,j\in[k]} x_{ij} = \sum_{i,j\in[k]} \frac{|M' \cap (C_i \times C_j)|}{|C_i|} =$$
$$\frac{1}{|C_1|} \sum_{i,j\in[k]} |M' \cap (C_i \times C_j)| = \frac{|M'|}{|C_1|} \geqslant \frac{k|M'|}{n} \overset{Eq.\ (3.4)}{\geqslant} \alpha k/2.$$

Now we apply Lemma 3.2.8 on fractional matching $\mathbf{x}$, for a parameter ε such that $\frac{\varepsilon^3}{12\ln(1/\varepsilon)} = 4\gamma$ which means $\varepsilon = \Theta((\gamma \log(1/\gamma))^{1/3})$. This results in a fractional matching $\mathbf{y}$ such that

(Y1) For any $i \in [k]$, $y_i \leqslant x_i$.

(Y2) $\text{supp}(\mathbf{y}) \subseteq \text{supp}(\mathbf{x})$.

(Y3) For any $i, j \in [k]$ either $y_{ij} = 0$ or $y_{ij} \geqslant 4\gamma$,

(Y4) $|\mathbf{y}| \geqslant |\mathbf{x}| - \varepsilon k$.

Choosing the constant in the definition of α to be large enough such that $\alpha/2 > \varepsilon$, we get from (X2) and (Y4) that $|\mathbf{y}| > 0$. Hence, there must exist some $y_{ij} \neq 0$. This by (Y2) implies that $x_{ij} \neq 0$ and so (C_i, C_j) must be a good pair. Additionally, $y_{ij} \neq 0$ implies by (Y3) that $y_i, y_j \geqslant y_{ij} \geqslant 4\gamma$ which by (Y1) also implies $x_i, x_j \geqslant 4\gamma$. From all of this, we get that there must be a good pair (C_i, C_j) such that at least 4γ fraction of each of C_i and C_j is matched by M', and there is an edge $(u, v) \in M'$ that goes from $u \in C_i$ to $v \in C_j$. Let us assume w.l.o.g. that $u \in A$, $v \in B$ (as we have not distinguished C_i, C_j in any other way up to now). Our next claim is that

(Z1) There are subsets $X \subseteq C_i \cap A$, $Y \subseteq C_j \cap B$ such that $|X| \geqslant \gamma|C_i|, |Y| \geqslant \gamma|C_j|$.

First, since C_i has a vertex in A that is matched in M', then it cannot be B-majority. Hence, $|C_i \cap V(M) \cap A| \geqslant |C_i \cap V(M)|/4 \geqslant |C_i \cap V(M')|/4 \geqslant \cdot 4\gamma|C_i|/4 = \gamma|C_i|$, where the third inequality follows from our earlier discussion that at least 4γ fraction of vertices in C_i are matched by M'. Similarly, since C_j has a vertex in B that is matched in M', it cannot be A-majority and thus $|C_j \cap V(M) \cap B| \geqslant |C_j \cap V(M)|/4 \geqslant |C_j \cap V(M')|/4 \geqslant \cdot 4\gamma|C_j|/4 = \gamma|C_j|$.

Now applying Claim 3.2.6 on these subsets X and Y of C_i and C_j in (Z1), proves that subsample F must include at least one edge between them, and so F is an α-hitting set. □

3.2.2. Second Step: From Hitting Set to Matching Cover

We now prove that any subgraph satisfying the hitting set requirement (Definition 3.2.3) is also a matching cover (Definition 3.2.1). This will follow from Hall's theorem (Proposition 2.2.2), and the proof similar to that of Lemma 9.3 in [GKK12].

Lemma 3.2.10 (From Hitting Set to Matching Cover)**.** *Let $G = (V, E)$ be any graph that is not necessarily bipartite. Then any subgraph H of G that is an α-hitting set is also an α-matching cover of G.*

Proof. Let H be a subgraph of G that is an α-hitting set. Consider any disjoint subsets $A, B \subset V$ with a maximum matching size of $\mu_G(A, B)$ in G. Let $P \subseteq A, Q \subseteq B$ be subsets with $|P| = |Q| = \mu_G(A, B)$ such that there exists a perfect matching M^* from P to Q. We show that there must exist a matching from P to Q in H of size $|P| - \alpha n$, i.e. $\mu_H(P, Q) \geqslant |P| - \alpha n$.

Suppose for the sake of contradiction that $\mu_H(P, Q) < |P| - \alpha n$. Then by applying the extended Hall's theorem (Proposition 2.2.2) to the bipartite subgraph of H induced by the bipartition (P, Q), there exists $P' \subset P$ such that $|P'| - |N_H(P') \cap Q| > \alpha n$. Now consider the edges of M^* that are incident on P' but *not* incident on $N_H(P')$, which themselves form a matching (call it M^{**}) of size $> \alpha n$. Now by the fact that H is an α-hitting set of G, there must be an edge in H connecting $V(M^{**}) \cap P$ to $V(M^{**}) \cap Q$. On the other hand, by the definition of M^{**}, we have $(V(M^{**}) \cap P) \subseteq P'$ but $V(M^{**}) \cap N_H(P') = \emptyset$, leading to a contradiction. □

We are now ready to prove Theorem 3.2.2.

Proof of Theorem 3.2.2. The output of Algorithm 3.2.1 being an $o(1)$-hitting set was proved in Lemma 3.2.9. By Lemma 3.2.10, the output subgraph is an $o(1)$-matching cover. This matching cover having at most $O(\gamma n^2) = n^2/(\log^* n)^{\Omega(1)}$ edges was proved in Claim 3.2.5. Finally, the running time follows from the algorithm of Proposition 2.2.3 for finding the regularity decomposition, and the fact that $Q(t, 1/\gamma) \leqslant \log n$ in Algorithm 3.2.1. □

3.3. A Single-Pass Streaming Algorithm

We prove Result 1 in this section. Our algorithm relies on using matching covers iteratively. We first present and prove a generic result that uses matching covers in a black-box way to obtain a streaming algorithm for finding matching covers and then extend it to obtain for Result 1.

3.3.1. A Streaming Algorithm for Matching Covers

We present an algorithm that computes the matching cover of a graph presented in a stream by iteratively computing matching covers of smaller subsets of the stream without losing "much" on the quality of the final matching cover. For technical reasons that will become clear later, we need this algorithm to work for multi-graphs as well.

Proposition 3.3.1. *For any integer $k \geqslant 1$ and any $\alpha \in (0, 1/10)$, there exists a single-pass streaming algorithm that computes an α-matching cover of n-vertex multi-graphs with at most m edges in space*

$$O\Big(\frac{m}{k} \cdot \log\left(\frac{n^2 \cdot k}{m}\right) + \mathsf{MC}(n, \alpha/2k) \cdot \log\left(\frac{n^2}{\mathsf{MC}(n, \alpha/2k)}\right) \cdot \log k\Big);$$

here, we assume we are given a subroutine `matching-cover` *that given adjacency matrix access to any n-vertex graph with m/k edges, can compute an $(\alpha/2k)$-matching cover with $\mathsf{MC}(n, \alpha/2k) \leqslant m/2k$ edges in $O((m/k) \cdot \log(n^2 \cdot k/m))$ space. The streaming algorithm requires calling the subroutine* `matching-cover` *$O(k)$ times and is deterministic as long as the* `matching-cover` *subroutine is deterministic.*

The algorithm in Proposition 3.3.1 is based on a novel use and modification of the widely used "Merge and Reduce" technique in the streaming literature (used previously e.g., for quantile estimation [MRL99, KLL16] or cut/spectral sparsifiers [McG14]). We give a high level overview of the algorithm here and present the formal description in Algorithm 3.3.1.

The algorithm maintains $t := O(\log k)$ different buffers $B_1, \ldots, B_t$ of edges throughout the stream (all these buffers store their edges using the succinct dynamic dictionary of Proposition 2.2.6 to save space). Buffer B_1 simply starts reading edges from the stream until it collects m/k edges; it will then use the (offline) subroutine `matching-cover` over these edges with parameter $\alpha' = \alpha/2k$ to

obtain an α'-matching cover of the subgraph of input on edges in B_1. Edges of this matching cover are then inserted to buffer B_2 and we empty buffer B_1, which will continue reading edges from the stream again. In the mean time, whenever buffer B_2 gets "full", this time meaning that it receives twice as many edges as $\mathsf{MC}(n, \alpha')$, we compute another α'-matching cover using `matching-cover`, this time over the edges in B_2, pass them to buffer B_3, and empty B_2 which continues receiving edges from buffer B_1. This process is done the same way across all buffers until all edges of the stream have passed (we prove buffer B_t never gets full so not having a buffer B_{t+1} is not a problem). At the end, we argue that the edges that are remained across all buffers $B_1, \ldots, B_t$ at the end of the stream form an α-matching cover of the input.

The analysis of the algorithm involves showing that: (i) fewer and fewer edges find their way to higher-indexed buffers, (ii) the repeated application of `matching-cover` does not blow up the approximation guarantee by too much, and (iii) all this can be implemented in a relatively small space. We now present the formal algorithm and its analysis.

Algorithm 3.3.1. An algorithm for Proposition 3.3.1.

Input: A multi-graph $G = (V, E)$ in the stream with n edges and at most m edges. We are also given integer $k \geqslant 1$ and approximation parameter $\alpha > 0$, and access to the (offline) subroutine `matching-cover` as specified in Proposition 3.3.1.

Output: An α-matching cover of G.

Parameters: We set $t := (\log k + 2)$ and $\alpha' := \alpha/2k$.

(i) Maintain the following **buffers** of edges $B_1, \ldots, B_t$ using succinct dynamic dictionary of Proposition 2.2.6 (we specify the details in Lemma 3.3.2):

- (a) Buffer B_1: add any arriving edge (u, v) arrives in the stream to B_1. Once size of B_1 reaches m/k, run `matching-cover` to find an α'-matching-cover of the subgraph (V, B_1) of G and add all those edges to B_2. Restart B_1 by deleting all its current edges.
- (b) Buffers B_i for $i > 1$: once size of B_i reaches $2 \cdot \mathsf{MC}(n, \alpha')$, run `matching-cover` to find an α'-matching-cover of the subgraph (V, B_i) of G and add all those edges to B_{i+1}[a]. Restart B_i by deleting all its current edges.

(ii) Return $(B_1 \cup \ldots \cup B_t)$ at the end of the stream.

[a] We will show in Claim 3.3.3 that this step never happens for buffer B_t, namely, it never gets "full", and thus the algorithm is well-defined even though there is no buffer B_{t+1}.

We start by analyzing the space complexity of Algorithm 3.3.1.

Lemma 3.3.2. *Algorithm 3.3.1 can be implemented in space of*

$$O\left(\frac{m}{k}\cdot\log\left(\frac{n^2\cdot k}{m}\right)+t\cdot\mathsf{MC}(n,\alpha')\cdot\log\left(\frac{n^2}{\mathsf{MC}(n,\alpha')}\right)\right).$$

Proof. At any point in the algorithm, B_1 contains $s_1 = O(m/k)$ edges and each B_i for $i > 1$ contains $s_i = O(\mathsf{MC}(n,\alpha'))$ edges. We can maintain each buffer B_i for $i \in [t]$ using a dedicated data structure $\mathbf{D}_i$ of Proposition 2.2.6 for the parameter s_i over the universe of all pairs of vertices:

- To add any edge (u,v) to a buffer B_i, we first check if (u,v) belongs to B_i via $\mathbf{D}_i.\,\texttt{member}(u,v)$, and if not use $\mathbf{D}_i.\,\texttt{insert}(u,v)$ to add the edge to B_i.
- To delete all edges from B_i, we simply erase $\mathbf{D}_i$ and start it from scratch.
- $\mathbf{D}_i$ now gives us an adjacency matrix access to the subgraph (V,B_i) by $\mathbf{D}_i.\,\texttt{member}(u,v)$ for finding if (u,v) is an edge in the subgraph.

Notice that even though G can be a multi-graph, each individual (V,B_i) is a simple graph.

By Proposition 2.2.6, the space needed for storing $\mathbf{D}_1$ and each $\mathbf{D}_i$ for $i > 1$ is, respectively,

$$(1+o(1))\cdot\log\binom{n^2}{m/k} = O(\frac{m}{k}\cdot\log\left(\frac{n^2\cdot k}{m}\right)),$$
$$(1+o(1))\cdot\log\binom{n^2}{\mathsf{MC}(n,\alpha')} = O(\mathsf{MC}(n,\alpha')\cdot\log(\frac{n^2}{\mathsf{MC}(n,\alpha')})).$$

Given there are $t-1$ buckets of the latter type, this bounds the space needed to store all the buffers in the algorithm as required in the lemma statement.

Finally, to implement each run of the subroutine `matching-cover`, since we have stored $\mathbf{D}_i$ for the buffer B_i, we can provide an adjacency matrix access to B_i for `matching-cover` (as required by the Proposition 3.3.1 statement), by simply checking $\mathbf{D}_i.\,\texttt{member}(u,v)$ for any query (u,v) to the adjacency matrix. As `matching-cover` is promised to use $O((m/k)\cdot\log(n^2\cdot k/m))$ space with this access, we get that the final bound on the space complexity of Algorithm 3.3.1. □

We now prove the correctness of Algorithm 3.3.1. To do so, we need the following definitions:

- Let $H_1^1,\ldots,H_{k_1}^1$ denote the k_1 separate matching covers constructed by the algorithm over

the edges of buffer B_1, one for each time that we restart B_1. Let $G^2 := H_1^1 \cup \ldots \cup H_{k_1}^1$ denote the graph that is sent to buffer B_2 throughout the algorithm (for notational convenience, we also define $G^1 = G$ as the input graph, namely, the graph that is sent to buffer B^1).

- For any $i \in [2 : t-1]$, similarly, let $H_1^i, \ldots, H_{k_i}^i$ denote the k_i separate matching covers constructed by the algorithm over the edges of buffer B_i. Let $G^{i+1} := H_1^i \cup \ldots \cup H_{k_i}^i$ denote the graph that is sent to buffer B_{i+1} throughout the algorithm.

We prove that the number of subgraphs at buffer B_i drops by a factor of 2^i compared to B_1.

Claim 3.3.3. *For any $i \in [t-1]$, $k_i \leqslant k/2^{i-1}$ and $k_t = 0$ meaning that bucket B_t never generates a matching cover (namely, it never gets full).*

Proof. We prove $k_i \leqslant k/2^{i-1}$ inductively. For the base case, since we restart buffer B_1 after each m/k edges in the stream and there are at most m edges in the stream, we have $k_1 \leqslant k$. For $i > 1$, the algorithm creates an α'-matching-cover H_j^i whenever bucket B_i gets full, which happens only when it collects $2 \cdot \mathsf{MC}(n, \alpha')$ edges. Moreover, the total number of edges ever sent to the bucket B_i is $|E(G^i)|$ by the definition of the subgraph G^i. Thus,

$$
\begin{aligned}
k_i &\leqslant \frac{1}{2 \cdot \mathsf{MC}(n,\alpha')} \cdot |E(G^i)| && \text{(as } k_i \text{ is equal to the number of times } B_i \text{ gets full)} \\
&\leqslant \frac{1}{2 \cdot \mathsf{MC}(n,\alpha')} \cdot \sum_{j=1}^{k_{i-1}} |E(H_j^{i-1})| && \text{(as } G^i \text{ is a union of } k_{i-1} \text{ matching-covers } H_j^{i-1}\text{)} \\
&\leqslant \frac{1}{2 \cdot \mathsf{MC}(n,\alpha')} \cdot k_{i-1} \cdot \mathsf{MC}(n,\alpha') \\
& && \text{(by the guarantee of } \texttt{matching-cover}\text{, } |E(H_j^{i-1})| \leqslant \mathsf{MC}(n,\alpha')\text{)} \\
&\leqslant \frac{k}{2^{i-1}},
\end{aligned}
$$

where the last step is by the induction hypothesis for $i-1$. This proves the first part of the claim. We now have that $k_{t-1} \leqslant k/2^{t-2} = k/2^{\log k} = 1$. Thus, only one matching-cover is ever sent to B_t and so B_t receives at most $\mathsf{MC}(n, \alpha')$ edges and never gets full. □

The following lemma captures the loss on the size of maximum matching that the algorithm maintains from one buffer to the next one. In other words, the cost we have to pay for introduction of

each level of buffers.

Lemma 3.3.4. *For any $i \in [t-1]$ and any disjoint subsets of vertices $X, Y \subseteq V$,*

$$\mu\left((G^{i+1} \cup B_i^f \cup \ldots \cup B_1^f)[X,Y])\right) \geqslant \mu\left((G^i \cup B_{i-1}^f \cup \ldots \cup B_1^f)[X,Y]\right) - k_i \cdot \alpha' \cdot n.$$

where B_j^f for $j \in [t]$ is the final content of the buffer at the end of the stream.

Proof. Fix any $i \in [t-1]$ and a maximum matching M_i^* of $(G^i \cup B_{i-1}^f \cup \ldots \cup B_1^f)[X,Y]$. We construct a matching M_{i+1} in $(G^{i+1} \cup B_i^f \cup \ldots \cup B_1^f)[X,Y]$ such that $|M_{i+1}| \geqslant |M_i^*| - k_i \cdot \delta \cdot n$. This will then immediately implies the lemma. To continue we need some more definition.

For any H_j^i for $j \in [k_i]$, let B_j^i denote the content of buffer B_i when the algorithm creates H_j^i. This way, H_j^i is a matching-cover of (V, B_j^i). Moreover, $B_1^i, \ldots, B_{k_i}^i$ together with B_i^f partition all the edges that are ever sent to buffer B_i, namely, the graph G^i. These edges are also further disjoint from $B_{i-1}^f, \ldots, B_1^f$ since the latter set of edges were never sent to buffer B_i. We can partition the edges of M_i^* between these sets and along the way define our matching M_{i+1} as well:

- For any $j \in [k_i]$, let $M_{i,j}^* := M_i^* \cap B_j^i$ and $M_{i,j}$ be the maximum matching in H_j^i between $X(M_{i,j}^*)$ and $Y(M_{i,j}^*)$.
- For any $i' \in [i]$, let $M_{i'}^{*,f} := M_i^* \cap B_i^f$ and $M_{i'}^f := M_{i'}^{*,f}$ which is between $X(M_{i'}^{*,f})$, $Y(M_{i'}^{*,f})$.
- Define $M_{i+1} := M_{i,1}^* \cup \cdots \cup M_{i,k_i}^* \cup M_i^f \cup \cdots \cup M_i^f$.

We note that M_{i+1} is a matching between X and Y because the sets of vertices $X(M_{i,j}^*)$ and $X(M_{i,j}^*)$ for $j \in [k_i]$, as well as $X(M_{i'}^{*,f})$ and $Y(M_{i'}^{*,f})$ for $i' \in [i]$ are all disjoint given they are defined with respect to a fixed matching M_i^* over disjoint sets of edges. Moreover, M_{i+1} belongs to $(G^{i+1} \cup B_i^f \cup \ldots \cup B_1^f)[X,Y]$ as H_j^i is part of G^{i+1} for $j \in [k_i]$. It thus only remains to bound the size of M_{i+1}.

For all $i' \in [i]$, $M^f_{i'}$ and $M^{*,f}_{i'}$ are the same so there is nothing to do here. For $j \in [k_i]$, we have,

$$|M_{i,j}| = \mu\left(H^i_j[X(M^*_{i,j}), Y(M^*_{i,j})]\right) \geqslant \mu\left(B^i_j[X(M^*_{i,j}), Y(M^*_{i,j})]\right) - \alpha' \cdot n$$

(as H^i_j is a α'-matching-cover of B^i_j and by Definition 3.2.1)

$$= |M^*_{i,j}| - \alpha \cdot n.$$

(as $M^*_{i,j}$ is a perfect matching in B^i_j between $X(M^*_{i,j})$ and $Y(M^*_{i,j})$)

Thus,

$$|M_{i+1}| = \sum_{j=1}^{k_i} |M_{i,j}| + \sum_{i'=1}^{i} |M^f_{i'}| \geqslant \sum_{j=1}^{k_i} (|M^*_{i,j}| - \alpha' \cdot n) + \sum_{i'=1}^{i} |M^{*,f}_{i'}| = |M^*_i| - k_i \cdot \alpha' \cdot n,$$

concluding the proof. □

We can now conclude the bound on the approximation ratio of the algorithm.

Lemma 3.3.5. *Algorithm 3.3.1 outputs an α-matching cover of any input multi-graph G.*

Proof. Recall that for every $i \in [t]$, B^f_i denotes the final content of the buffer B_i. Moreover by Claim 3.3.3, buffer B_t never gets full and thus $B^f_t = G^t$. Finally, the algorithm returns $H := (B^f_1, \ldots, B^f_t)$. Fix any disjoint sets of vertices $X, Y \subseteq V(G)$. We have,

$$\begin{aligned}
\mu(H[X,Y]) &= \mu\left((B^f_t \cup B^f_{t-1} \cup \ldots \cup B^f_1)[X,Y]\right) && \text{(by the definition of } H\text{)} \\
&= \mu\left((G^t \cup B^f_{t-1} \cup \ldots \cup B^f_1)[X,Y]\right) && \text{(as } B^f_t = G^t\text{)} \\
&\geqslant \mu\left((G^{t-1} \cup B^f_{t-2} \cup \ldots \cup B^f_1)[X,Y]\right) - k_{t-1} \cdot \alpha' \cdot n && \text{(by Lemma 3.3.4 for } i = t-1\text{)} \\
&\geqslant \mu(G[X,Y]) - \sum_{i=1}^{t-1} k_i \cdot \alpha' \cdot n \\
& && \text{(by repeatedly applying Lemma 3.3.4 for all } i < t-1 \text{ and since } G^1 = G\text{)} \\
&\geqslant \mu(G) - \sum_{i=1}^{t-1} (k/2^{i-1}) \cdot \alpha' \cdot n && \text{(by Claim 3.3.3, } k_i \leqslant k/2^{i-1}\text{)} \\
&\geqslant \mu(G) - 2k \cdot \alpha' \cdot n && \text{(by the sum of the geometric series)} \\
&= \mu(G) - \alpha \cdot n. && \text{(by the choice of } \alpha' = \alpha/2k\text{)}
\end{aligned}$$

This implies that for every disjoint subsets of vertices $X, Y \subseteq V(G)$, we have $\mu(H[X,Y]) \geqslant \mu(G[X,Y]) - \alpha \cdot n$, thus making H an α-matching cover of G by Definition 3.2.1. □

Proof of Proposition 3.3.1. The space complexity of the algorithm follows from Lemma 3.3.2 by plugging the value of $\alpha' = \alpha/2k$ and $t = \log k + 1$. The correctness follows from Lemma 3.3.5. Finally, Algorithm 3.3.1 is deterministic modulo any potential randomness used by `matching-cover`.

□

3.3.2. A Streaming Matching Algorithm via Regularity Lemma

We now use Proposition 3.3.1 together with our Theorem 3.2.2 to formalize Result 1 as follows.

Theorem 3.3.6 (Formalization of Result 1)**.** *There is a randomized single-pass streaming algorithm that with high probability computes a $(1 - o(1))$-approximate matching of a graph presented in a stream with adversarial order of edge arrivals in $n^2/(\log^* n)^{\Omega(1)}$ space and polynomial time.*

Proof. To apply Proposition 3.3.1, we need a subroutine `matching-cover` for computing an $(\alpha/2k)$-matching cover (for parameters α and k to be determined soon) on any n-vertex graph with n^2/k edges. Theorem 3.2.2 provides such an algorithm with parameters

$$(\alpha/2k) = \frac{1}{(\log^* n)^{\delta_1}} \quad \text{and} \quad \mathsf{MC}(n, \alpha/2k) = \frac{n^2}{(\log^* n)^{\delta_2}},$$

for some absolute constants $\delta_1, \delta_2 \in (0,1)$. Let $\alpha = 1/(\log^* n)^{3\delta_1/4}$ and $k = \frac{1}{2} \cdot (\log^* n)^{\delta_1/4}$, which satisfies the conditions above. Moreover, by Proposition 2.2.4, we can implement Algorithm 3.2.1 of Theorem 3.2.2 in polynomial time and space $O((n^2/k) \cdot \log k) = n^2/(\log^* n)^{\Omega(1)}$, given only adjacency matrix access to its input graph. This way, by Proposition 3.3.1, we obtain a single-pass streaming algorithm that with high probability computes an α-matching cover in space $n^2/(\log^* n)^{\Omega(1)}$.

The main algorithm in the theorem is as follows. We store the first $2n^2/k$ edges in the stream using succinct dynamic dictionary of Proposition 2.2.6 in $n^2/(\log^* n)^{\Omega(1)}$ space. In parallel, we also run the algorithm mentioned above to obtain an α-matching cover of G. The space complexity and polynomial runtime of the algorithm is thus already established.

We now prove the correctness. If $\mu(G) \leqslant n/k$, then by Fact 2.2.1, we have stored all edges of the graph and thus at the end can simply return a maximum matching of the stored edges; to do so, we simply run Hopcroft-Karp algorithm [HK73] by providing it with the adjacency matrix of the stored edges using `member` query on the succinct dynamic dictionary (which only requires $O(n \log n)$ additional space beside the input). Thus, in this case, we obtain an exact maximum matching of the input graph.

If $\mu(G) > n/k$, then we can pick X and Y in the definition of matching cover output by the algorithm of Proposition 3.3.1 to be the endpoints of the maximum matching of G, and have,

$$\mu(H) \geqslant \mu(G) - \alpha \cdot n \geqslant (1 - \alpha \cdot k) \cdot \mu(G) = (1 - 1/(\log^* n)^{\delta_1/2})\mu(G),$$

which is $(1 - o(1)) \cdot \mu(G)$ as desired. This concludes the proof. □

CHAPTER 4

FIRST ALGORITHMS AND LOWER BOUNDS FOR WEIGHTED GRAPH SPARSIFICATION IN THE LINEAR SKETCHING AND TURNSTILE STREAMING MODELS

In this chapter, we initiate the study of weighted graph sparsification in the linear sketching and turnstile streaming models, and present the first set of algorithms and lower bounds in such models. The results from this chapter are based on my work [CKL22b] joint with Chen and Khanna.

We start with Section 4.1 which gives an overview of our weighted cut and spectral sparsification algorithms in these two models. Next, Section 4.2 gives an overview of our lower bound for weighted spectral sparsification by linear sketching. Then Sections 4.3 and 4.4 present in detail our algorithms and their analysis for weighted cut and spectral sparsification, respectively.

The remaining Sections 4.5-4.10 are devoted to proving our lower bounds for weighted spectral sparsification, which relies on a number of new tools for analyzing certain *matrix-weighted* graphs. Section 4.5 first defines such matrix-weighted graphs and sets up some notation. Then Sections 4.6, 4.7, 4.8, 4.9 present the new tools we develop for analyzing them. Finally, Section 4.10 proves our lower bounds for weighted spectral sparsification.

4.1. Overview of weighted cut and spectral sparsification algorithms

We now give an overview of our weighted cut and spectral sparsification algorithms.

4.1.1. Overview of the algorithm for weighted cut sparsification

We will first give a recap of how the unweighted cut sparsification algorithm works in [AGM12b], and then describe what new ideas are needed to extend it to work for weighted graphs.

Recap of the unweighted cut sparsification algorithm in [AGM12b]. At a high-level, the approach taken is to reduce cut sparsification to (repeatedly) recovering a spanning forest of a subgraph of the input graph, obtained by sampling edges *uniformly* at some rate $p \in (0, 1)$ known beforehand. This task is then further reduced to the task of sampling an edge connecting S to $\bar{S}$ for an arbitrary subset S of vertices, as this can be used to create a spanning forest by growing

connected components.

Now to implement this latter task, in the sketching phase, we apply an ℓ_0-sampler sketch to the incidence vector of each vertex u (i.e. each column of the edge-vertex incidence matrix) in the *sub-sampled graph*. Then in the recovery phase, in order to recover an edge going out of a vertex set S, we add up the sketches of the vertices inside S. By linearity, this summed sketch is taken over the sum of the incidence vectors of vertices inside S, and the latter contains exactly the edges going out of S, since the edges inside cancel out. As a result, we can recover an edge going out of S, and create a spanning forest of the sub-sampled graph. Note that this approach crucially utilizes the fact that the edges are sampled uniformly at a rate that is known *beforehand*. This means that we can sample all $\binom{n}{2}$ edge slots beforehand, and apply the linear sketch only to the sampled edge slots.

Our approach for weighted cut sparsification. We also reduce the task to recovering a spanning forest in a sub-sampled graph. However, the latter graph is now obtained by sampling edges *non-uniformly*. Specifically, we need to recover a spanning forest in a subgraph obtained by sampling each edge e with probability $\min\{w_e p, 1\}$ for some parameter $p \in (0,1)$ that is known beforehand. Therefore, in order to apply the idea as in the unweighted case, we will now need to design a variant of ℓ_0-sampler that, given a vector $x \in \mathbb{R}^N_{\geqslant 0}$ and a parameter $p \in (0,1)$, recovers a nonzero entry of x after each entry $i \in [N]$ is sampled with probability $\min\{x_i p, 1\}$. We call such a sampler "weighted edge sampler".

Note that the edge weights are *not* known to us beforehand, so we cannot sample the edge slots with our desired probabilities as in the unweighted graph case. We instead build such a weighted edge sampler using a *rejection sampling* process, in which we sample edges uniformly at $\widetilde{O}(1)$ geometric rates, but use ℓ_1-*samplers* to try recovering edges at each rate, and only output a recovered edge e if the sampling rate $\approx w_e p$. We then show that with high probability, we can efficiently find a desired edge.

Roughly, our analysis involves proving that there exists a geometric rate q such that, after uniformly sampling edges at rate q, the total weight of edges e satisfying $w_e p \approx q$ accounts for a large portion of that of all sampled edges. As a result, by using a few independent ℓ_1-samplers, we can find one

such edge with high probability.

4.1.2. Overview of the algorithm for weighted spectral sparsification

As in previous linear sketches for unweighted graphs [KLM+14, KMM+20], the key task is to recover edges with $\tilde{\Omega}(1)$ effective resistances (or in weighted case, $\tilde{\Omega}(1)$-leverage scores), which we refer to as *heavy edges*. The high-level idea used in previous works is to (i) compute, for each vertex pair s, t, a set of vertex potentials $x_{s,t} \in \mathbb{R}^n$ induced by an electrical flow from s to t, and then (ii) apply an ℓ_2-heavy hitter to $B_G x_{s,t} \in \mathbb{R}^{\binom{n}{2}}$ to try recovering the edge (s, t), where B_G is the edge-vertex incidence matrix of G (see Section 2.4.2). They achieve (i) by simulating an iterative refinement process in [LMP13]. To achieve (ii), they make a key observation that

$$\|B_G x_{s,t}\|_2^2 = x_{s,t}^T B_G^T B_G x_{s,t} = x_{s,t}^T L_G x_{s,t}$$

is the energy of $x_{s,t}$, and the entry of $B_G x_{s,t}$ indexed by edge (s, t) is $(B_G x_{s,t})_{(s,t)} = b_{s,t}^T x = x_s - x_t$. Therefore by the energy minimization characterization of effective resistances (Fact 2.4.5), whenever the effective resistance between s, t is $b_{s,t}^T L_G^\dagger b_{s,t} \geqslant \tilde{\Omega}(1)$, we have

$$(B_G x_{s,t})_{(s,t)}^2 \geqslant \tilde{\Omega}(1) \|B_G x_{s,t}\|_2^2,$$

and hence the entry (s, t) is an ℓ_2-heavy hitter.

However, when the graph is weighted, we are only allowed to access the graph through linear measurements on its weight vector w_G. As a result, we can only apply ℓ_2-heavy hitters to $W_G B_G x_{s,t}$, whose squared ℓ_2-norm is $x_{s,t}^T B_G^T W_G^2 B_G x_{s,t}$. Now notice that $B_G^T W_G^2 B_G$ is the Laplacian matrix of a "squared" graph (call it G^{sq}), which has the same edges as G, but whose edges are weighted by w_e^2 as opposed to w_e. Therefore, we will be recovering edges that are heavy in G^{sq} instead of in G if we apply the same approach as in previous works. Unfortunately, a heavy edge in G is not in general heavy in G^{sq}, since the energy on the edges with very large weights will blow up when we square the edge weights (i.e. $w_e^2(x_u - x_v)^2 \gg w_e(x_u - x_v)^2$), and hence make the total energy grow unboundedly.

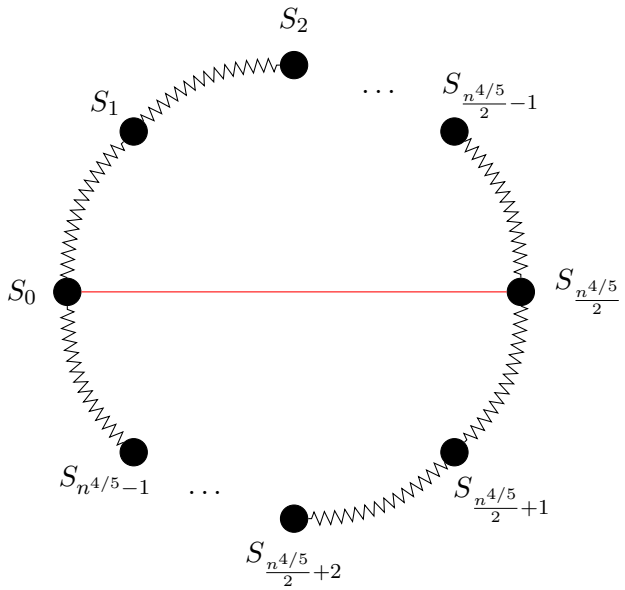

Figure 4.1: A block cycle graph on n vertices. Each S_i represents a block of $n^{1/5}$ vertices connected by a clique and each zigzag represents the edges of a complete bipartite graph between adjacent blocks. The red edge represents the crossing edge. All edges along the cycle have weights $n^{2/5}$, and the crossing edge has weight 1.

To see an intuitive example, suppose G is a "block cycle graph" on n vertices whose edges are generated as follows (see also Figure 4.1):

1. Partition the vertices into $n^{4/5}$ blocks $S_0, \ldots, S_{n^{4/5}-1}$, each with $n^{1/5}$ vertices.
2. For each $0 \leqslant i < n^{4/5}$, add on S_i a complete graph of $n^{1/5}$ vertices with edge weights $n^{2/5}$, i.e. $n^{2/5}K_{n^{1/5}}$.
3. For each $0 \leqslant i < n^{4/5}$, add on (S_i, S_{i+1}) a complete bipartite graph of $2n^{1/5}$ vertices with edge weights $n^{2/5}$ and bipartition (S_i, S_{i+1})[6], i.e. $n^{2/5}K_{n^{1/5},n^{1/5}}$.
4. Finally, add a "crossing edge" e^* of weight 1 between a randomly chosen vertex pair s, t.

We note that, in this construction, the crossing edge e^* spans $\Omega(n^{4/5})$ consecutive blocks, Note that, typically, the crossing edge e^* spans $\Omega(n^{4/5})$ consecutive blocks, and therefore has effective resistance (and also leverage score) $\Omega(1)$.

Proposition 4.1.1. *If $s \in S_i, t \in S_j$ such that $\min\left\{|i-j|, n^{4/5} - |i-j|\right\} \geqslant \Omega(n^{4/5})$, then the*

[6]Note that we consider $i+1$ as 0 when $i = n^{4/5} - 1$.

effective resistance of e^* satisfies $r_{e^*} \geqslant \Omega(1)$.

Proof. Let s, t be the endpoints of the crossing edge e^*. By the energy minimization characterization of effective resistances (Fact 2.4.5), it suffices to show that there is a set of vertex potentials whose normalized energy with respect to s, t is $O(1)$. Specifically, consider the set of potentials $x \in \mathbb{R}^n$ such that $x_u = \frac{i}{n^{4/5}}$ for all $u \in S_i$. Then we have $x_s - x_t = \Theta(1)$, and the total energy is $\sum_{e=(u,v)} w_e(x_u - x_v)^2 = n^{6/5} \cdot n^{2/5}(\Theta(n^{-4/5}))^2 + \Theta(1) = \Theta(1)$, as desired. □

However, in the squared graph G^{sq}, all edge weights along the cycle are blown up by a factor of $n^{2/5}$, and thus e^* only has leverage score $O(n^{-2/5})$ in G^{sq}. To recover in a vector x every entry with ℓ_2-contribution $\geqslant n^{-2/5} \|x\|_2^2$, one will need an $\Omega(n^{-2/5})$ factor blowup in the number of linear measurements, resulting in a total of $n^{7/5}$ measurements needed to recover e^*.

We can in fact improve the number of linear measurements needed for recovering e^* to $\widetilde{O}(n^{6/5})$ using the *vertex sampling* trick, an idea first used in [FKN21] for sketching spanners. Namely, consider sampling a vertex set $C \subset V$ by including each vertex with probability $n^{-1/5}/100$, and looking at the vertex-induced subgraph $G^{\text{sq}}[C]$. Then one can show that, conditioned on $e^* \in G^{\text{sq}}[C]$, with constant probability, the two endpoints of e^* will be disconnected in $G^{\text{sq}}[C] \setminus e^*$. As a result, the leverage score of e^* becomes 1 in $G^{\text{sq}}[C]$, and we can recover e^* by recovering heavy edges in $G^{\text{sq}}[C]$, which, as will show, can be done using $\widetilde{O}(|C|) \approx \widetilde{O}(n^{4/5})$ measurements. Since $e^* \in G[C]$ with probability $\approx \frac{1}{n^{2/5}}$, repeating this sampling process independently for $\widetilde{O}(n^{2/5})$ times allows us to recover e^* in at least one vertex induced subgraph. This results in a linear sketch of $\widetilde{O}(n^{6/5})$ measurements.

What if we slightly increase each block's size to $n^{1/5+\delta}$ and decrease the edge weights along the cycle to $n^{2/5-3\delta}$? While one can still verify that the crossing edge e^* has leverage score $\Omega(1)$, applying the same vertex sampling process as above will not disconnect the endpoints of e^* with $\Omega(1)$ probability. However, one can alternatively show that, with constant probability, the number of edges along the cycle reduces by a factor of $n^{2/5}$. Since now the energy of each edge only blows up by a factor smaller than $n^{2/5}$ in G^{sq}, this will also make the leverage score of e^* become $\Omega(1)$ in $G^{\text{sq}}[C]$, and thus we can apply the same linear sketch of $\widetilde{O}(n^{6/5})$ measurements.

The above warm-up seems to suggest that the sampling rate of $\approx n^{-1/5}$ is a sweet spot for recovering heavy edges in any graphs with the block cycle structure. Indeed, we prove a key vertex sampling lemma showing that in *any* weighted graph G, a heavy edge e in G is also likely heavy in a vertex-induced subgraph of G^{sq} obtained by sampling vertices at rate $\approx n^{-1/5}$. This is proved by carefully analyzing the structures of the edges of different weights after vertex sampling, and then explicitly constructing a set of vertex potentials with small total energy in the induced subgraph. Finally, by integrating this lemma into an iterative refinement process in [LMP13] (as the authors did in [KLM$^+$14, KMM$^+$20]) and a spectral sparsification algorithm in [Kou14], we are able to recover a spectral sparsifier of G using $\widetilde{O}(n^{6/5})$ linear measurements. We note that the latter step of using heavy edge recovery to build a spectral sparsifier is also more involved than in the unweighted case and requires a few extra techniques; we refer the reader to the overview at the beginning of Section 4.4.3 and the discussion therein for more details.

We note that this method of recovering heavy edges by vertex sampling is inspired by the one used in [KLM$^+$14] for spanners. However, for spectral sparsification, the correctness of such a method follows from fairly different reasoning, and the proof is arguably more involved.

4.2. Overview of lower bound for weighted spectral sparsification

In this section we give an overview of our lower bound for weighted spectral sparsification (Theorem 6). We prove our lower bound on a family of hard instances that turn out to have the exact same structure as the one in Figure 4.1, which we used to illustrate the difficulty of recovering spectral sparsifiers in weighted graphs. Specifically, our hard instances are weighted "block cycle graphs" plus an extra *crossing edge* that is included with probability 1/2. In a block cycle graph, the vertices are partitioned into blocks that are arranged in a cyclic manner. Each block is a complete graph, and the vertices of adjacent blocks are connected by a complete bipartite graph. Here we draw all edge weights from Gaussian distributions and permute the vertices uniformly at random. On such graphs, for a suitable choice of edge weights, computing a spectral sparsifier essentially boils down to detecting the presence/absence of the crossing edge. We then show that for the latter task, the success probability of any incidence sketch can be bounded by the "effective resistance" of the crossing edge in a *matrix-weighted* graph, where the matrix weights are in turn determined by

the sketching matrix.

We note that this family of hard instances has a similar structure to the ones conjectured in [FKN21]. However, instead of using Bernoulli distributions on the edges as suggested in [FKN21], we use Gaussian distributions, which makes it easier to build the connection to effective resistances.

In order to show that the effective resistance is small for any incidence sketch with a limited number of measurements, we develop a number of new tools for analyzing such matrix-weighted graphs. Most importantly, we present (i) a matrix-weighted analog of the widely used expander decomposition of ordinary graphs [GR99, KVV04, ST04], and (ii) a proof that a random vertex-induced subgraph of a matrix-weighted expander is also an expander with high probability. We highly recommend reading Section 4.2.3 to get intuition on why these two techniques are useful, where we use the ordinary graph version of (i) and (ii) to prove a lower bound for a simple class of sketches.

The rest of this section is structured as follows. In Section 4.2.1 we describe the distribution from which we generate our hard instances. In Section 4.2.2 we explain how we bound the success probability of an incidence sketch by the effective resistance in a matrix-weighted graph. In Section 4.2.3 we prove, as a warm-up, a lower bound for a simple class of sketches, where we only need to analyze the effective resistances in *ordinary graphs*. Finally in Section 4.2.4 we outline our proof for arbitrary incidence sketches, which requires analyzing matrix-weighted graphs. We note that some proofs in this section are deferred to Appendix B.2.

4.2.1. The hard distribution

We first state how we generate the input weighted graph $G = (V, E, w)$. Let n be the number of vertices and define $s \stackrel{\text{def}}{=} n^{1/5}$ and $\ell \stackrel{\text{def}}{=} n^{4/5}$. We choose a random permutation $\pi : 1..n \to 1..n$ and construct a block cycle graph as follows. The i-th block (where $0 \leqslant i < \ell$) consists of vertices $\pi(si+1), \ldots, \pi(si+s)$. For simplicity we denote the a-th vertex in the i-th block (i.e. $\pi(si+a)$) as $u_{i,a}$. The block index i will always be modulo ℓ implicitly. We then add a complete graph to each block, and a complete bipartite graph between each pair of adjacent blocks. Namely, for each $0 \leqslant i < \ell$, we add a graph G_i with edges connecting $u_{i,a}, u_{i,b}$ for all $a < b \in \{1, \ldots, s\}$, and add

another bipartite graph $G_{i,i+1}$ with edges connecting $u_{i,a}, u_{i+1,b}$ for all $a, b \in \{1, \ldots, s\}$. Finally, with probability $1/2$, we add an edge between vertices $\pi(1)$ and $\pi(n/2+1)$ (assume n is even). We refer to this edge as the *crossing edge* with respect to π and any other edge in G as a *non-crossing edge* with respect to π. We will omit "with respect to π" when the underlying permutation π is clear.

We next describe how the edge weights are determined. The weights of all non-crossing edges are drawn independently from $\mathcal{N}(8n^{2/5}, n^{4/5}\log^{-1} n)$ (the Gaussian distribution with mean $8n^{2/5}$ and variance $n^{4/5}\log^{-1} n$). The weight of the crossing edge is drawn from the standard Gaussian $\mathcal{N}(0,1)$. If the crossing edge has negative weight, we say the input is *invalid*, and accept any sketch as a valid sketch. Our goal will be to detect the presence/absence of the crossing edge with high probability.

In the following, we will call the conditional distribution on the presence of the crossing edge the Yes *distribution*, and call the conditional distribution on the absence of the crossing edge the No *distribution*. We then show that with high probability, the effective resistance of the crossing edge is large, and therefore any linear sketch for computing spectral sparsifiers must distinguish between the two distributions with good probability.

Proposition 4.2.1. *With probability at least $1 - 1/n$, all non-crossing edges have weights in the range $[4n^{2/5}, 12n^{2/5}]$, and as a result the effective resistance between vertices $\pi(1)$ and $\pi(n/2+1)$ is at least $1/48$ in a* No *instance.*

Proposition 4.2.2. *Any linear sketch that can compute a 1.0001-spectral sparsifier with probability 0.9 can distinguish between the* Yes *and* No *distributions with probability 0.6.*

The first proposition follows from an application of the Chernoff bound (Theorem 2.3.11). The proof of the second proposition is deferred to Appendix B.2.

In the following, we will assume, for ease of our analysis, that the sketch will be given the permutation π *after* computing the linear sketch. That is, the recovery algorithm $\mathcal{A}$ takes as input both Φw and π. We will show that even with this extra piece of information, any incidence sketch with $n^{21/20-\varepsilon}$ measurements for constant $\varepsilon > 0$ cannot distinguish between the Yes and No distributions with high

probability.

4.2.2. A bound on the success probability via effective resistance

We first show that for our lower bound instance, any incidence sketch can be reduced to a more restricted class of linear sketches by only increasing the number of measurements by an $O(\log n)$ factor. Specifically, let us fix an arbitrary orientation of the edges, and consider sketches taken over the weighted signed edge-vertex incidence matrix $B^w \in \mathbb{R}^{\binom{n}{2} \times n}$, where the latter is given by

$$B^w_{eu} = \begin{cases} w_e & e \in E \text{ and } u \text{ is } e\text{'s head} \\ -w_e & e \in E \text{ and } u \text{ is } e\text{'s tail} \\ 0 & \text{otherwise.} \end{cases}$$

That is, the algorithm must choose a (random) sketching matrix $\Phi \in \mathbb{R}^{k \times \binom{n}{2}}$ with the e^{th} column $\phi_e \in \mathbb{R}^k$ corresponding to the edge slot e. The sketch obtained is then $\Phi B^w \in \mathbb{R}^{k \times n}$. Notice that the total number of measurements in ΦB^w is kn, as each vertex applies the sketching matrix Φ to its incident edges. Let us call this class of sketch *signed sketches*. By Yao's minimax principle [Yao77], to prove a lower bound for distinguishing the Yes and No distributions, it suffices to focus on deterministic sketches. The proof of the proposition below appears in Appendix B.2.

Proposition 4.2.3 (Reduction to signed sketches)**.** *Consider any incidence sketch of N measurements with a deterministic sketching matrix $\Phi \in \mathbb{R}^{N \times \binom{n}{2}}$ and a recovery algorithm $\mathcal{A}$ that, given Φw and π, distinguishes between the* Yes *and* No *distributions with probability at least* 0.6*. Then there exists a signed sketch with a sketching matrix $\Phi' \in \mathbb{R}^{k \times \binom{n}{2}}$, where $k = O(1) \cdot \max\{1, \frac{N \log n}{n}\}$, and a recovery algorithm $\mathcal{A}'$ that, given $\Phi' B^w$ and π, distinguishes between the* Yes *and* No *distributions with probability at least* 0.55*.*

Let us now fix a sketching matrix $\Phi \in \mathbb{R}^{k \times \binom{n}{2}}$ and aim to obtain an upper bound on the success probability of any signed sketch using Φ. For notational convenience, let us write $(\Phi B^w)_{\text{yes}}$ to denote ΦB^w conditioned on the presence of the crossing edge and $(\Phi B^w)_{\text{no}}$ to denote ΦB^w conditioned on the absence of the crossing edge. We will also write $(\Phi B^w)_{\pi,\text{yes}}$ or $(\Phi B^w)_{\pi,\text{no}}$ to denote an extra conditioning on the permutation being π in addition to the presence/absence of the crossing edge.

Then to bound the success probability of any signed sketch using Φ , it suffices to show that the total variation distance (TV-distance) between $(\Phi B^w)_{\text{yes}}$ and $(\Phi B^w)_{\text{no}}$ is small.

To state our upper bound on the TV-distance, we need to first introduce some notation. For an edge (u, v), define $b_{uv} \in \mathbb{R}^{nk}$ by writing it as a block vector (with block size k) as follows:

$$b_{uv} = \begin{pmatrix} 0 \\ \vdots \\ \phi_{uv} \\ 0 \\ \vdots \\ -\phi_{uv} \\ 0 \\ \vdots \end{pmatrix} \begin{matrix} \\ \\ u^{\text{th}} \text{ block} \\ \\ \\ v^{\text{th}} \text{ block} \\ \\ \\ \end{matrix} \in \mathbb{R}^{nk}, \tag{4.1}$$

where $\phi_{uv} \in \mathbb{R}^k$ is the column of Φ corresponding to the edge slot (u, v). For a permutation π, we then define $L_\pi = \sum_{\text{non-crossing } (u,v)} n^{4/5} \log^{-1} n \, b_{uv} b_{uv}^T$. The following proposition is essentially a consequence of Theorem 2.3.10 [DMR18], which bounds the TV-distance between multivariate Gaussians with the same mean. We give its proof in Appendix B.2. Note that we use $\dagger$ to denote taking the Moore-Penrose pseudoinverse of a matrix.

Proposition 4.2.4. *For any permutation π such that $b_{\pi(1)\pi(n/2+1)}$ is in the range[7] of L_π,*

$$d_{\text{TV}}((\Phi B^w)_{\pi,\text{yes}}, (\Phi B^w)_{\pi,\text{no}}) \leqslant O(1) \cdot \min\left\{1, b_{\pi(1)\pi(n/2+1)} L_\pi^\dagger b_{\pi(1)\pi(n/2+1)}\right\}.$$

Our plan is then to show that $b_{\pi(1)\pi(n/2+1)} L_\pi^\dagger b_{\pi(1)\pi(n/2+1)}$ is small on average for every choice of a signed sketch with $k = n^{1/20-\varepsilon}$ for constant $\varepsilon > 0$.

Note that if $k = 1$ and each $\phi_{uv} \in \{0, 1\}$, then L_π is exactly the Laplacian matrix of the graph (call it $\mathcal{H}_\pi$) that is formed by the non-crossing edges (u, v) such that $\phi_{uv} = 1$, where each edge is

[7] Recall that the range of a symmetric matrix is the linear span of its columns.

weighted by $n^{4/5}\log^{-1} n$. Therefore, $b_{\pi(1)\pi(n/2+1)}L_\pi^\dagger b_{\pi(1)\pi(n/2+1)}$ is the effective resistance between $\pi(1)$ and $\pi(n/2+1)$ in $\mathcal{H}_\pi$, if $\phi_{\pi(1)\pi(n/2+1)} \neq 0$ (otherwise $b_{\pi(1)\pi(n/2+1)}$ is the zero vector).

In fact, to get a quick intuition as to why we should expect the effective resistance between $\pi(1)$ and $\pi(n/2+1)$ to be small, let us assume $\phi_{uv} = 1$ for all edge slots (u, v). Then, for any permutation π, $\mathcal{H}_\pi$ is the graph formed by *all* non-crossing edges, each weighted by $n^{4/5}\log^{-1} n$. Note that these weights are about $n^{2/5}$ times larger than the weights $\Theta(n^{2/5})$ in the actual input graph (Proposition 4.2.1). As a result, the effective resistance between $\pi(1)$ and $\pi(n/2+1)$ is about $n^{2/5}$ times smaller than the effective resistance between them in the input graph (the former roughly equals $n^{-2/5}$).

When $k > 1$, we can view L as the Laplacian of a *matrix-weighted* graph (again, call it $\mathcal{H}_\pi$) formed by the non-crossing edges, where each edge (u, v) has a $k \times k$ matrix weight $n^{4/5}\log^{-1} n\, \phi_{uv}\phi_{uv}^T$. Now $b_{\pi(1)\pi(n/2+1)}L_\pi^\dagger b_{\pi(1)\pi(n/2+1)}$ can be seen as the (generalized) effective resistance between $\pi(1)$ and $\pi(n/2+1)$ in $\mathcal{H}_\pi$.

4.2.3. Warm-up: one-row signed sketches have small TV-distance

As a warm-up, we show that for any signed sketch, in the case that $k = 1$ and the sketching matrix Φ has 0/1 entries, we have, for any constant $\varepsilon > 0$,

$$\mathbb{E}_\pi\left[d_{\mathrm{TV}}\left((\Phi B^w)_{\pi,\mathrm{yes}}, (\Phi B^w)_{\pi,\mathrm{no}}\right)\right] \leqslant \frac{1}{n^{1/5-O(\varepsilon)}}. \tag{4.2}$$

By Proposition 4.2.4, we know that $d_{\mathrm{TV}}\left((\Phi B^w)_{\pi,\mathrm{yes}}, (\Phi B^w)_{\pi,\mathrm{no}}\right)$ can be bounded by the effective resistance between $\pi(1), \pi(n/2+1)$ in $\mathcal{H}_\pi$ if $\phi_{\pi(1)\pi(n/2+1)} = 1$, and is zero otherwise. Here $\mathcal{H}_\pi$ is formed by the non-crossing edges whose $\phi_{uv} = 1$, where each edge (u, v) has scalar weight $n^{4/5}\log^{-1} n$. We can focus on the Φ's whose number of nonzero entries is at least $n^{9/5+\varepsilon}$, since otherwise

$$\Pr_\pi\left[\phi_{\pi(1)\pi(n/2+1)} = 1\right] = \frac{\mathrm{nnz}(\Phi)}{\binom{n}{2}} \leqslant \frac{1}{n^{1/5-O(\varepsilon)}},$$

and we would already have our desired result (4.2).

Our proof of (4.2) will rely on decomposing $\mathcal{H}_\pi$ into expanders with large minimum degree. Since $\mathcal{H}_\pi$'s edges all have the same weight $n^{4/5}\log^{-1} n$, it is more convenient to work with the *unweighted version of* $\mathcal{H}_\pi$, which we denote by H_π. We now briefly review the definition of unweighted expanders, as well as state a known expander decomposition lemma that we will utilize.

Definition 4.2.5 (Expander)**.** An unweighted graph $H = (V, E)$ is a ζ-expander for some $\zeta \in [0, 1]$ if its conductance is at least ζ, namely, for every nonempty $S \subset V$, we have

$$|E(S, V - S)| \geqslant \zeta \cdot \min\{\mathrm{vol}(S), \mathrm{vol}(V - S)\},$$

where $\mathrm{vol}(S)$ is the total degree of vertices in S.

Note that in the lemma below, we slightly abuse the notion of "regular graphs". Specifically, we will say a graph is regular if its minimum vertex degree $d_{\min}$ is not much smaller than the average degree d.

Lemma 4.2.6 (Almost regular expander decomposition, see e.g. [KKTY21])**.** *Given an unweighted graph $H = (V, E)$ with average degree $d \geqslant 16$, there exists a subgraph $I = (U, F)$ where $U \subseteq V$ and $F \subseteq E$ such that I is a $\frac{1}{16\log n}$-expander with minimum degree $d_{\min} \geqslant \frac{d}{16}$.*

We will also need the following lemma, which shows that a random vertex-induced subgraph of an expander with large minimum degree is almost certainly an expander. We give the proof of this lemma in Section 4.9.1. To the best of our knowledge, even this result was not known before.

Lemma 4.2.7 (Expanders are preserved under vertex sampling)**.** *There exists a $\theta = \theta(n) = n^{o(1)}$ with the following property. Consider an unweighted $\frac{1}{16\log n}$-expander $H = (V, E)$ with minimum degree $d_{\min} \geqslant 4 \cdot 10^6 \cdot \theta(n)$. For any $s \geqslant \frac{4\cdot 10^6}{d_{\min}} \cdot \theta(n) \cdot n$, let $C \subseteq V$ be a uniformly random vertex subset of size s. Then with probability at least $1 - 1/n^7$, the vertex-induced subgraph $H[C]$ is a $\frac{1}{n^{o(1)}}$-expander with minimum degree at least $\frac{s}{2n} \cdot d_{\min}$.*

Proof of (4.2) using Lemmas 4.2.6,4.2.7. As argued above we can assume w.l.o.g. that $\mathrm{nnz}(\Phi) \geqslant n^{9/5+\varepsilon}$. We want to obtain, for each edge slot e satisfying $\phi_e = 1$, conditioned on e being the crossing edge w.r.t. π, an upper bound (call it u_e) on the typical effective resistance between the endpoints

of e in the graph $\mathcal{H}_\pi$. In other words, conditioned on e being the crossing edge, u_e should be an upper bound on the effective resistance between the endpoints of e in $\mathcal{H}_\pi$ with high probability over π. Then the total variation distance between $(\Phi B^w)_{\text{yes}}$ and $(\Phi B^w)_{\text{no}}$ can be bounded by

$$\mathbb{E}_\pi \left[d_{\text{TV}} \left((\Phi B^w)_{\pi,\text{yes}}, (\Phi B^w)_{\pi,\text{no}} \right) \right] \leqslant O(1) \cdot \frac{1}{\binom{n}{2}} \sum_{e:\phi_e=1} u_e. \tag{4.3}$$

To obtain the u_e's, let us define the unweighted graph $H_\phi = (V, E_\phi)$ where E_ϕ contains *all* edges e whose $\phi_e = 1$ (including the ones not present in the input graph, i.e. $|E_\phi| = \text{nnz}(\Phi)$). Now consider the following process, where we repeatedly delete an expander subgraph from H_ϕ and obtain u_e's for the edges in the expander.

1. While $|E_\phi| \geqslant 10^9 n^{9/5+\varepsilon}$:
   (a) Find a subgraph $I = (U, F)$ of $H_\phi = (V, E_\phi)$ that is a $\frac{1}{16 \log n}$-expander with minimum degree $d_{\min} \geqslant \frac{|E_\phi|}{8n}$ (existence is guaranteed by Lemma 4.2.6).
   (b) For each edge $f \in F$, let $u_f \leftarrow \left(\frac{10^9 n^{9/5+\varepsilon}}{|E_\phi|} \right)^2$.
   (c) Delete the edges in F from H_ϕ by letting $E_\phi \leftarrow E_\phi \setminus F$.
2. Let $u_f \leftarrow 1$ for all f in the remaining E_ϕ.

To show that u_e's are valid upper bounds, let us consider a fixed iteration of the while loop. For $i = 0, \ldots, n^{4/5} - 1$, let U_i denote the vertices in I that are in the i^{th} block of the input block cycle graph:

$$U_i \stackrel{\text{def}}{=} U \cap \left\{ \pi(n^{1/5} i + 1), \ldots, \pi(n^{1/5} i + n^{1/5}) \right\}.$$

Then by Chernoff bounds, with probability at least $1 - 1/n^5$ over the random choice of π, we have $|U_i| \geqslant \frac{|U|}{2n^{4/5}} \geqslant \frac{4 \cdot 10^6}{d_{\min}} \cdot |U|^{1+\varepsilon}$. Then by invoking Lemma 4.2.7, with probability at least $1 - 1/n^4$ over π, all vertex-induced subgraphs $I[U_i \cup U_{i+1}]$ are $\frac{1}{n^{o(1)}}$-expanders with minimum degree at least $\frac{|E_\phi|}{16 n^{9/5}}$. Using this fact, we obtain the following claim, whose proof appears in Appendix B.2.

Claim 4.2.8. *For each edge $f \in F$, conditioned on f being the crossing edge, with probability at*

least $1 - 1/n^2$ *over* π*, the effective resistance between the endpoints of* f *in* $\mathcal{H}_\pi$ *is at most* u_f*.*

Now let us divide the above process for obtaining u_e's into $O(\log n)$ phases, where in phase $i \in \{1, \ldots, O(\log n)\}$, we have $|E_\phi| \in (\binom{n}{2}/2^i, \binom{n}{2}/2^{i-1}]$. Then we have

$$\sum_{e:\phi_e=1} u_e = \sum_{e:\phi_e=1} \left(\frac{10^9 n^{9/5+\varepsilon}}{|E_\phi|}\right)^2 \leqslant n^{O(\varepsilon)} \cdot \sum_{i=1}^{O(\log n)} \frac{\binom{n}{2}}{2^i} \cdot \left(\frac{2^i}{n^{1/5}}\right)^2$$
$$\leqslant n^{8/5+O(\varepsilon)} \sum_{1}^{O(\log n)} 2^i \leqslant n^{9/5+O(\varepsilon)}$$

where in the first line we have used $n^{O(\varepsilon)}$ to hide the constant factors, and the last inequality holds since in the last phase we have $2^i \leqslant n^{1/5}$. Plugging this into (4.3) finishes the proof. □

4.2.4. The general case: proof of Theorem 6

Note that even though for $k = 1$, the TV-distance is $\widetilde{O}(n^{-1/5})$, this does not imply that k must be large for the TV-distance to become $\Omega(1)$.

By Proposition 4.2.3, in order to prove Theorem 6, it suffices to prove the following:

Theorem 4.2.9. *For any fixed sketching matrix* $\Phi \in \mathbb{R}^{k \times \binom{n}{2}}$ *where* $k \leqslant n^{1/20-\varepsilon}$ *for some constant* $\varepsilon > 0$*, we have*

$$\mathbb{E}_\pi \left[d_{\mathrm{TV}} \left((\Phi B^w)_{\pi,\mathrm{yes}}, (\Phi B^w)_{\pi,\mathrm{no}} \right) \right] \leqslant o(1).$$

By Proposition 4.2.4, our goal is to bound the "effective resistance" $b_{\pi(1)\pi(n/2+1)} L_\pi^\dagger b_{\pi(1)\pi(n/2+1)}$ between vertices $\pi(1), \pi(n/2+1)$ in the matrix-weighted graph $\mathcal{H}_\pi$ consisting of the non-crossing edges, where edge (u, v) has matrix weight $n^{4/5} \log^{-1} n \, \phi_{uv} \phi_{uv}^T \in \mathbb{R}^{k \times k}$. We will do so by (significantly) generalizing our previous approach based on expander decomposition for ordinary graphs in Section 4.2.3. Our approach for the $k = 1$ case essentially consists of two steps: (i) decomposing the graph H_ϕ into large expander subgraphs and (ii) proving that a random vertex induced subgraph of an expander is still an expander.

First note that there does not appear to be a combinatorial analog of conductance in matrix-weighted

graphs, which suggests that we should define expanders in an algebraic way. Let us first recall the algebraic characterization of expanders for ordinary, unweighted graphs. The definition is based on eigenvalues of the *normalized Laplacian* of the graph, which is given by $N = D^{-1/2}LD^{-1/2}$, where D is a diagonal matrix with D_{uu} equal to the degree d_u of u.

Definition 4.2.10 (Algebraic definition of ordinary, unweighted expanders)**.** An unweighted graph H is a ζ-expander for some $\zeta \in [0, 1]$ if the smallest nonzero eigenvalue of its normalized Laplacian matrix N is at least ζ.

By Cheeger's inequality [AM85], for $\zeta \geqslant \tilde{\Omega}(1)$, this definition translates to that the graph H is a union of vertex-disjoint combinatorial expanders, each with conductance $\tilde{\Omega}(1)$. To come up with an analogous definition for matrix-weighted graphs, let us first define their associated matrices formally.

Matrices associated with matrix-weighted graphs. We consider a $k \times k$ matrix-weighted graph $H = (V, E)$ with $|V| = n$. For each edge $(u, v) \in E$, there is a vector $\phi_{uv} \in \mathbb{R}^k$, indicating that (u, v) is weighted by the $k \times k$ rank-1 matrix $\phi_{uv}\phi_{uv}^T$.

Definition 4.2.11 (Degree matrices)**.** For a vertex u, its generalized degree is given by

$$D_u = \sum_{u \sim v} \phi_{uv}\phi_{uv}^T \in \mathbb{R}^{k \times k}.$$

We then define the $nk \times nk$ degree matrix D as a block diagonal matrix (with block size $k \times k$), with the u^{th} block on the diagonal being $D_{uu} = D_u \in \mathbb{R}^{k \times k}$.

Definition 4.2.12 (Laplacian matrices)**.** The Laplacian matrix is given by $L = \sum_{(u,v) \in E} b_{uv}b_{uv}^T$, where b_{uv}'s are defined in (4.1).

We will call b_{uv} the *incidence vector* of edge (u, v). Note that the Laplacian matrix here differs from the connection Laplacian matrix [KLP$^+$16] which is also defined to be a block matrix. In particular, the Laplacian matrix of a matrix-weighted graph is not necessarily block diagonally dominant (bDD) (Definition 1.1 of [KLP$^+$16]).

Definition 4.2.13 (Normalized Laplacian matrices)**.** The normalized Laplacian matrix is given by $N \stackrel{\text{def}}{=} D^{\dagger/2}LD^{\dagger/2}$. Equivalently, we have $N = \sum_{(u,v) \in E} D^{\dagger/2}b_{uv}b_{uv}^T D^{\dagger/2}$.

We will call $D^{\dagger/2}b_{uv}$ the *normalized incidence vector* of edge (u, v).

The following proposition says that similar to scalar-weighted graphs, the eigenvalues of the normalized Laplacian of a matrix-weighted graph are also between $[0, 2]$. The proof this proposition appears in Appendix B.3 (in "Proof of Proposition 4.5.16").

Proposition 4.2.14. *The eigenvalues of N are between $[0, 2]$.*

Now, a first attempt might be to define matrix-weighted expanders to also be graphs whose normalized Laplacians' nonzero eigenvalues are large, and then try to decompose any matrix-weighted graph into large expander subgraphs. However, we show that the latter goal may not be achievable in general, by presenting in Appendix B.1 a hard instance, for which any large subgraph has a small nonzero eigenvalue.

Our approach. In light of the hard instance, we loosen the requirement of being an expander by allowing small eigenvalues, but requiring instead that each edge, compared to the average, does not have too large "contribution" to the small eigenvectors. Formally, we want that every edge's normalized incidence vector has small (weighted) projection onto the bottom eigenspace. We will also need an analog of "almost regularity", which for ordinary, unweighted graphs says that the minimum degree is large. We give the formal definition of an almost regular matrix-weighted expander below.

Definition 4.2.15 (Almost regular matrix-weighted expanders)**.** For a $k \times k$ matrix-weighted graph H, let $\lambda_1 \leqslant \ldots \leqslant \lambda_{nk}$ be the eigenvalues of its normalized Laplacian N, and let $f_1, \ldots, f_{nk} \in \mathbb{R}^{nk}$ be a set of corresponding orthonormal eigenvectors. We say H is a (γ, ζ, ψ)-almost regular expander if

1. (γ-almost regularity) For every vertex u and every incident edge $(u, v) \in E$, we have

$$\phi_{uv}^T D_u^\dagger \phi_{uv} \leqslant \frac{\gamma \cdot k}{n}. \tag{4.4}$$

2. ((ζ, ψ)-expander) For every edge $(u, v) \in E$ we have

$$\left(D^{\dagger/2}b_{uv}\right)^T \left(\sum_{i:\lambda_i \in (0,\zeta]} \frac{1}{\lambda_i} f_i f_i^T\right) D^{\dagger/2}b_{uv} \leqslant \frac{\psi \cdot k^2}{n^2}. \tag{4.5}$$

The LHS of (4.4) is the so-called *leverage score* of ϕ_{uv} w.r.t. D_u (Definition 2.3.5). It is known that the sum of leverage scores equals the rank of the matrix:

Proposition 4.2.16. *For any fixed vertex u, $\sum_{(u,v)\in E} \phi_{uv}^T D_u^\dagger \phi_{uv} = \operatorname{rank}(D_u)$.*

Since D_u is a $k \times k$ matrix, we have $\operatorname{rank}(D_u) \leqslant k$. Therefore, in the case that u has $\Omega(n)$ incident edges, (4.4) is essentially saying that no incident edge's leverage score exceeds the average by too much.

To get intuition for condition (4.5), we need the following two results. The first theorem is proved in Section 4.6. The second proposition is proved in Appendix B.2.

Theorem 4.2.17. *Let H be a $k\times k$-matrix weighted graph that is γ-almost regular (in the sense of (4.4)). Then for any $\zeta \in (0,1)$, the number of eigenvalues of its normalized Laplacian that are between $(0,\zeta]$ is at most $\frac{\gamma\cdot k^2}{(1-\zeta)^2}$.*

Proposition 4.2.18. *Let ℓ be the number of λ_i's that are between $(0,\zeta]$. Then*

$$\sum_{(u,v)\in E} \left(D^{\dagger/2} b_{uv}\right)^T \left(\sum_{i:\lambda_i\in(0,\zeta]} \frac{1}{\lambda_i} f_i f_i^T\right) D^{\dagger/2} b_{uv} = \ell. \tag{4.6}$$

Therefore, in the case that $|E| = \Omega(n^2)$, (4.5) is essentially saying that the LHS for every edge (u,v) does not exceed the average by too much.

We then show that every dense enough matrix-weighted graph can indeed be made into an expander by downscaling a small number of edges. To this end, let us define, for a *scaling* $s : E \to [0,1]$, the rescaled graph H^s, which is obtained from H by rescaling each edge (u,v)'s weight to $s_{uv}^2 \phi_{uv}\phi_{uv}^T$. The proof of the following theorem appears in Sections 4.7 and 4.8.

Theorem 4.2.19. *There is an algorithm that, given any $k \times k$ matrix-weighted graph $H = (V,E)$ with $|E| \geqslant \Omega(n^2)$, outputs a scaling $s : E \to [0,1]$ such that*

1. *The rescaled graph H^s is a (γ,ζ,ψ)-almost regular expander for*

$$\gamma = 8\log n,\ \zeta = \frac{1}{\log n},\ \psi = 16k^2\log^3 n.$$

2. *The number of edges* $(u, v) \in E$ *with* $s_{uv} < 1$ *is* $o(n^2)$.

We next show that almost-regular expanders are preserved under vertex sampling. However, we will now use a different notion of "preservation". To state our specific result, let us define some additional notations. For a vertex subset $C \subseteq V$, we write $L_{G[C]}$ to denote the Laplacian of the vertex-induced subgraph $G[C]$. We also let D_{CC} be the submatrix of D (the degree matrix of the original graph H) with rows and columns restricted to vertices in C, and let $(f_i)_C$ denote, for an eigenvector f_i, the vector f_i with indices restricted to C. We then have the following theorem, whose proof appears in Section 4.9.2.

Theorem 4.2.20. *There exists a* $\theta = \theta(n) \leqslant n^{o(1)}$ *with the following property. Let* $H = (V, E)$ *be a* $k \times k$ *matrix-weighted,* (γ, ζ, ψ)*-almost regular expander where* $\zeta \leqslant 1/\log n$. *For an* $s \geqslant 2 \cdot 10^6 \gamma\psi\zeta^{-1}k^2\theta(n)$, *let* $C \subseteq V$ *be a uniformly random vertex subset of size* s. *Then with probability at least* $1 - 1/n^5$, *we have that*

1. *The null space of* $D_{CC}^{\dagger/2} L_{G[C]} D_{CC}^{\dagger/2}$ *is exactly the linear span of* $\{(f_i)_C : \lambda_i = 0\}$.
2. *For all vectors* $x \in \mathbb{R}^{|C|k}$ *such that* $x^T (f_i)_C = 0, \forall i : \lambda_i = 0$,

$$x^T \left(D_{CC}^{\dagger/2} L_{G[C]} D_{CC}^{\dagger/2} \right)^\dagger x \leqslant n^{o(1)} \cdot x^T \left(\frac{n^2}{s^2} \sum_{i:\lambda_i \in (0,\zeta]} \frac{1}{\lambda_i} (f_i)_C (f_i)_C^T + \frac{n}{s} \cdot \frac{1}{\zeta} I \right) x. \tag{4.7}$$

We argue that (4.7) is roughly saying that the pseudoinverse of the subgraph $G[C]$ can be bounded by the pseudoinverse of the original graph that is (i) restricted to indices in C and (ii) rescaled in a certain way. For technical reasons, on the LHS of (4.7) we normalize the Laplacian of the vertex-induced subgraph using the degree matrix of the *original graph* H. As for the RHS, we can see it as a rescaled version of the pseudoinverse of N restricted to C, by noting that

$$\left(N^\dagger \right)_{CC} = \sum_{\lambda_i > 0} \frac{1}{\lambda_i} (f_i)_C (f_i)_C^T.$$

Thus, on the RHS of (4.7) we blow up the small eigenvalues quadratically in 1/(sampling rate), but blow up the large eigenvalues linearly in 1/(sampling rate).

With these tools, we are finally able to prove Theorem 6. We present the proof in Section 4.10.

4.2.4.1 Techniques for proving Theorems 4.2.17, 4.2.19, 4.2.20

We now explain, at a very high level, the techniques that we use to prove these three key theorems, as well as their connections to previous works. More details can be found in the subsequent sections.

Proof of Theorem 4.2.17. We consider the "spectral embedding" induced by the bottom eigenvectors of the normalized Laplacian, which maps each vertex to a rectangular matrix. Such a spectral embedding may be seen as the matrix-weighted counterpart of the ones for scalar-weighted graphs, which map each vertex to a *vector*. The latter embeddings were previously used to prove higher-order Cheeger inequalities [LRTV12, LGT14]. We show that, for matrix-weighted graphs that are almost regular (in the sense of (4.4)), the spectral embedding has vertex-wise bounded spectral norm (Lemma 4.6.4), and as a result the number of bottom eigenvectors must be small (Theorem 4.6.2).

Proof of Theorem 4.2.19. Our proof consists of two steps: (i) decomposing the graph into an almost regular graph (Theorem 4.7.1), and (ii) decomposing the graph into an almost regular expander (Theorem 4.8.2). In achieving (ii), we actually invoke (i) repeatedly to maintain the almost regularity of the graph.

As noted above, the almost regularity condition (4.4) is essentially saying that no incident edge has leverage score too large comparing to the average. A similar task to (i) has in fact been investigated by a previous work [CLM$^+$15], where the authors showed that given a set of vectors, one can, by downscaling a small number of them, make every vector have small leverage score comparing to the average. This result is achieved by an algorithm that iteratively downscales vectors with large leverage scores, while analyzing how each vector's leverage score changes in the process. While it is possible to directly invoke the result from [CLM$^+$15] to get a large almost regular graph, its guarantee does not suffice for our purpose of smoothly incorporating (i) into (ii). In particular, since we will repeatedly invoke (i) in (ii), we need, in addition to that the number of rescaled edges is small, an extra bound on the number of completely deleted edges (i.e. those rescaled to 0) that is proportional to the rank change of the degree matrix D. As a result, we design a more involved algorithm for obtaining the scaling as well as carry out a more careful analysis of the algorithm.

Achieving (ii) turns out to be much more challenging. Although the LHS of (4.5) may be seen as a leverage score, there is the intrinsic difficulty that whenever the edge weights change, so do the eigenvalues and eigenvectors of the normalized Laplacian, as well as the degree matrix itself (hence also the normalized incidence vectors). Thus it is not clear how the LHS of (4.5) will change. As a result, when trying to obtain a desired scaling, we have to use some global measure of progress. This is in contrast to (i), where we can track the leverage score change of each edge *locally*. We resolve this issue by considering, as a potential function, the determinant of the normalized Laplacian restricted to the bottom eigenspace. In other words, our potential function is the product of the eigenvalues of N that are between $(0, \zeta]$[8]. We show that, by a delicate global analysis of such a potential function, we are able to make the graph an expander by only downscaling a small number of edges.

Proof of Theorem 4.2.20. Our proof is motivated by the *approximate Gaussian elimination* of the Laplacian matrices of scalar-weighted graphs, which was previously used as an algorithmic tool for solving graph structured linear systems [KLP$^+$16, KS16] and building data structures for dynamically maintaining effective resistances [DKP$^+$17, LZ18, LPYZ20]. Our approach also relies on analyzing matrix-valued martingales which played a key role in [KS16]. which have played key roles in constructing vertex/subspace sparsifiers [KS16, LS18, FGL$^+$21].

Let us first briefly review the Gaussian elimination of the Laplacian matrix of a scalar-weighted graph. Roughly speaking, by eliminating the row and column of L corresponding to a vertex u, we can obtain another Laplacian matrix L' supported on $V \setminus \{u\}$ whose pseudoinverse *equals* the pseudoinverse of the original L restricted to $V \setminus \{u\}$ (i.e. $(L')^\dagger = (L^\dagger)_{V\setminus\{u\}V\setminus\{u\}}$). Given a vertex subset $C \subseteq V$, one can also eliminate the vertices outside of C one by one and get a Laplacian matrix L'' supported on C with the same property that $(L'')^\dagger = (L^\dagger)_{CC}$. The matrix L'' is referred to as the *Schur complement* of L onto C. However, the graphs associated with L' and L'' could be dense, which are inefficient for algorithm design. Therefore [KS16] showed that one can perform an *approximate* Gaussian elimination by, upon each elimination, implicitly sub-sampling the edges in L'. They then showed that we eventually get a good approximation to L'' by analyzing a matrix-valued martingale induced by this process.

[8] Due to technical reasons, the actual potential function slightly differs from the one stated here.

We now explain how to apply this idea to prove Theorem 4.2.20. Since we are considering an induced subgraph $G[C]$ where C is a uniformly random subset of size s, we can also view the process for choosing C as deleting a sequence of $n - s$ vertices from V uniformly at random. Our goal is to compare the pseudoinverse of $G[C]$ with that of the original graph, therefore it suffices to compare it with the Schur complement of L onto C. We will in fact do such a comparison upon the elimination of every vertex. That is, if we let C_i be the set of remaining vertices at the i^{th} step, then we want to compare the Laplacian of $G[C_i]$ with the Schur complement of L onto C_i. At a high level, we do so by setting up a matrix-valued martingale, and show that it has good concentration when G is a matrix-weighted expander (in the sense of Definition 4.2.15).

4.3. A linear sketching algorithm for weighted cut sparsification

In this section, we present a linear sketch with $\tilde{O}(n\varepsilon^{-3})$ measurements that computes a $(1+\varepsilon)$-cut sparsifier of a weighted graph. In particular, our linear sketch will be an incidence sketch. Our algorithm is obtained by generalizing the cut sparsifier algorithm of [RSW18], which is an $O(\log n)$ round cut query algorithm that works only for unweighted graphs. We first describe the weighted generalization of the algorithm of [RSW18] in a model-oblivious manner, and then show how to implement it by linear sketching.

4.3.1. A model oblivious algorithm for weighted cut sparsification

We present a model-oblivious algorithm `weighted-cut-sparsify` for weighted cut sparsification in Figure 4.2, which generalizes the idea presented in [RSW18] for unweighted graphs. Here, we assume the edge weights of the input graph are between $[1, U]$ for some $U \geqslant 1$ that is known to us. We characterize the performance of the algorithm in the lemma below.

Lemma 4.3.1. *Fix $\alpha = 800, \beta = 400, \gamma = 100$ at Line 2 of the algorithm `weighted-cut-sparsify`. Let ε be an arbitrary number in $(0, 1)$. Then, the algorithm `weighted-cut-sparsify` outputs, with high probability, a $(1+\varepsilon)$-cut sparsifier of the input graph G with $O(n\varepsilon^{-2}\log n)$ edges.*

To prove the proposition, we will need the following lemma.

Lemma 4.3.2. *In each iteration ℓ, with high probability:*

1. *The edges in $\tilde{G}$ that survive the contractions have strength at most $2^{\ell-1}$.*

2. The edges in $\tilde{G}$ that are within the components $C_1, \ldots, C_r$ have strength $[2^{\ell-4}, 2^{\ell}]$.

Proof. We show that at the end of each iteration ℓ, the edges within $C_1, \ldots, C_r$ have strength $\geqslant 2^{\ell-4}$, and the edges between them have strength at most $2^{\ell-1}$. Then by noting that all edges in the graph have strength at most nU, the lemma follows by an induction on ℓ.

Consider iteration ℓ of the for loop. Let $V_1, \ldots, V_t$ be the partition of $\tilde{G}$ into maximal $2^{\ell-4}$-strongly connected components. Then by Fact 2.4.1 there exists a way to arrive at these components by starting from the entire graph $\tilde{G}$ and iteratively removing a cut with size $< 2^{\ell-4}$. By applying a Chernoff bound and a union bound over the sequence of (at most $n-1$) cuts that we remove in this process, we have that after sampling, each of these cuts has size at most $50 \log n$. As a result, the partition $C_1, \ldots, C_r$ of G_ℓ into maximal $100 \log n$-strongly connected components is a refinement of $V_1, \ldots, V_t$. This implies that all edges within $C_1, \ldots, C_r$ have strength $\geqslant 2^{\ell-4}$.

Now consider the partition $V_1, \ldots, V_s$ of $\tilde{G}$ into maximal $2^{\ell-1}$-strongly connected components. After sampling at Line 3a, these components still have min cut $\geqslant 100 \log n$ with high probability. Thus, the partition $C_1, \ldots, C_r$ of G_ℓ into maximal $100 \log n$-strongly connected components is a coarsening of $V_1, \ldots, V_s$. This implies that all edges going across different C_i's have strength at most $2^{\ell-1}$. This finishes the proof of the lemma. □

Proof of Lemma 4.3.1. By Lemma 4.3.2, whenever we sample an edge e of the graph, we sample it with probability $p_e \geqslant \min\left\{1, 50 w_e k_e^{-1} \varepsilon^{-2} \log n\right\}$ and re-weight it to w_e/p. Moreover, in the last iteration $\ell = 0$, all edges are within the components $C_1, \ldots, C_r$ (since each edge's strength is at least 1). Therefore ultimately all edges get sampled in our algorithm. As a result, we get with high probability a $(1+\varepsilon)$-cut sparsifier. On the other hand, the probability p_e with which we sample e also satisfies $p_e \leqslant 800 w_e k_e^{-1} \varepsilon^{-2} \log n$, and thus we get a sparsifier with $O(n\varepsilon^{-2} \log n)$ edges. □

4.3.2. Implementation by linear sketching

Now we show how to implement the algorithm by linear sketching. The implementation is motivated by the techniques first used in [AGM12a]. Note that it suffices to implement the two edge sampling processes at Lines 3a, 3(b)iii and the contraction operations at Line 3(b)iv.

$H \leftarrow$ `weighted-cut-sparsify`(G, ε)

1. Initially, let the sparsifier H be an empty graph, and let $\tilde{G} \leftarrow G$.
2. Fix some sufficiently large constants $\alpha > \beta > \gamma > 0$.
3. For $\ell \in \{\log(nU), \log(nU) - 1, \ldots, 1, 0\}$:
   (a) Obtain G_ℓ from $\tilde{G}$ by keeping each edge w.p. $p_e := \min\left\{\beta \cdot w_e \cdot 2^{-\ell} \log n, 1\right\}$ and with weight $(\beta \cdot w_e \cdot 2^{-\ell} \log n)/p_e$.
   (b) In each connected component of G_ℓ:
      i. While there exists a cut of weight $\leqslant \gamma \cdot \log n$, remove the edges in that cut and recurse on both sides; repeat until there is no such cut.
      ii. Let $C_1, \ldots, C_r$ be the connected components induced by the remaining edges.
      iii. For each edge e in $\tilde{G}$ with endpoints in the same C_i, add it to H with probability $p_e := \min\left\{\alpha \cdot w_e \cdot \varepsilon^{-2} 2^{-\ell} \log n, 1\right\}$ and with weight w_e/p_e.
      iv. Update $\tilde{G}$ by contracting each of $C_1, \ldots, C_r$.

Figure 4.2: Model oblivious algorithm for weighted cut sparsification.

We note that the implementation of both sampling processes can be seen as the following task: we first independently generate a uniformly random real number $R_e \in [0, 1)$ for each edge slot e, and then recover all edges satisfying $R_e < w_e p$ for some given p (which in iteration ℓ equals $\beta 2^{-\ell} \log n$ for the first process and $\alpha\varepsilon^{-2} 2^{-\ell} \log n$ for the second process). Here we can generate the R_e's offline, but have to recover the sampled edges using linear sketching. We achieve the latter by repeatedly finding a spanning forest formed by the sampled edges. We will show that the sampled edges can be found by performing $\widetilde{O}(1)$ iterations of spanning forest recovery.

We shall first show how to recover a spanning forest formed by the sampled edges via linear sketching. To this end, we need a linear sketching subroutine that we call *weighted edge sampler*, with the following guarantee.

Lemma 4.3.3. *Let $R_1, \ldots, R_N$ be N numbers independently and uniformly at random generated from $[0, 1)$, let $c > 0$ be an arbitrary constant, and let $p \in (0, 1)$ be a parameter. There exists a linear sketch of $\mathrm{polylog}(N, 1/p)$ measurements with the following guarantee. For any vector $w \in \mathbb{R}^N$, if there exists an entry e such that $R_e \leq w_e p$ and $w_e > N^{-c}$, then with high probability, the sketch recovers an index e' such that $R_{e'} \leqslant (1 + \varepsilon) w_{e'} p$ along with a $(1 + \varepsilon)$-approximate estimate of $w_{e'}$.*

To find a spanning forest of the sampled edges (those with $R_e < w_e p$), we apply the weighted edge sampler sketch to the incidence vectors of the vertices in G. Specifically, we fix an arbitrary

orientation of each of the $\binom{n}{2}$ potential edges. Then for a vertex $u \in G$, we consider its incidence vector $b_u \in \mathbb{R}^{\binom{n}{2}}$ given by

$$(b_u)_e = \begin{cases} w_e & e \in G, \, u \text{ is } e\text{'s head} \\ -w_e & e \in G, \, v \text{ is } e\text{'s tail} \\ 0 & e \notin G \text{ or } e \text{ does not touch } u. \end{cases} \tag{4.8}$$

Let $t = \text{polylog}(n)$. For any $i \in [t]$, let A_i be an independently generated weighted edge sampler sketching matrix. For each $i \in [t]$ and vertex $u \in G$, we compute the sketch $A_i b_u$. Thus we make $\widetilde{O}(n)$ measurements in total. Now using the sketches we have taken, we recover a spanning forest of sampled edges via a $\widetilde{O}(1)$-round process as follows.

In the first round, for each vertex u, we find an arbitrary outgoing edge using the weighted edge sampler sketch $A_1 b_u$. We then find all connected components induced by these edges, and add up the sketches $A_2 b_u$ of vertices within the same component. Note that the edges within the same component cancel out in the summation, so the resulting sketches are in fact taken over the outgoing edges of each component. As a result, in the next round we are able to find an outgoing edge of each component. We then proceed similarly in the i^{th} round using sketches $A_i b_u$'s. Since in each round, the number of components is at least reduced by a factor of 2, we can find a spanning forest in $O(\log n)$ rounds of this process.

In order to iteratively find $\widetilde{O}(1)$ edge-disjoint spanning forests, each time we find one, we "delete" the found edges from the other linear sketches, and restart the $O(\log n)$-round process above. Note that however, since we do not have the exact weights of the edges (Lemma 4.3.3 only gives approximation of them), we do not delete the found edges completely, bur rather decrease each of their weights by an $\Omega(1)$ factor.

Finally, to implement the contraction operations at Line 3(b)iv, we once again add the sketches of the vertices within each contracted component, just as we did in finding spanning forests. Then starting from the next iteration, the sketches work for the contracted graph $\tilde{G}$.

We conclude this subsection by proving that the sampled edges can be recovered by $\widetilde{O}(1)$ edge-

disjoint spanning forests.

Lemma 4.3.4. *The edges in G_ℓ can be found by $O(\log^2 n)$ edge-disjoint spanning forests.*

Proof. We first show that the edges in G_ℓ all have low strength.

Claim 4.3.5. *Every edge in the graph G_ℓ has strength $O(\log n)$.*

Proof. By Lemma 4.3.2, we know that at the beginning of iteration ℓ, all edges in $\tilde{G}$ have strength at most 2^ℓ. This means that there is a way of removing all edges in the graph by iteratively removing a cut of size $\leqslant 2^\ell$. Then after sampling at Line 3a, these cuts all have size $O(\log n)$. Thus it follows that all edges in G_ℓ have strength $O(\log n)$. □

Let $C > 0$ be a constant such that all edges in G_ℓ have strength $\leqslant C \log n$. Then this claim implies that G_ℓ is *uniformly sparse*, in the sense that for any vertex induced subgraph $G[S]$ where $S \subseteq V$, the number of edges in $G[S]$ is $(|S|-1)C \log n$. Indeed, each edge in $G[S]$ has strength only smaller than in G, and thus all edges in $G[S]$ can be removed by iteratively (for at most $|S|-1$ times) removing a cut of size $C \log n$. This in particular means that at any point, a spanning forest contains an $1/(C \log n)$ fraction of the total remaining edges. Thus $O(\log^2 n)$ edge-disjoint spanning forests recover all edges in G_ℓ. □

4.3.3. Proof of Lemma 4.3.3

Roughly, we will simulate the non-uniform sampling process, where we want to sample each e with probability $w_e p$, by sampling the elements uniformly, but at different geometric rates. We will then essentially implement a *rejection sampling* process. Specifically, when subsampling all elements at some uniform rate q, we use ℓ_1-samplers to recover a few elements that are sampled. We will then check if any one of the sampled elements e satisfies $w_e p \approx q$. If so, we will output this element; otherwise, we go the next sampling rate and repeat this step. We will show that with high probability, we will successfully recover a desired element.

Proof of Lemma 4.3.3. We will analyze the linear sketch given in Figure 4.3. The basic idea of the algorithm is as follows: for each $0 \leq j \leq \gamma$, we maintain $\text{polylog}(N, \frac{1}{p})$ number of γ_1-sketches given

$e \leftarrow$ `weighted-edge-sampler`$(w \in \mathbb{R}^N, p, R_1, \ldots, R_N)$

- Let $t = 10^4 c\gamma \log N$ where γ is the minimum integer such that $2^{-\gamma} < pN^{-c}$.
- For each $1 \le i \le t$ and each $0 \le j \le \gamma$, independently generate an γ_1-sampling matrix A_i^j.
- For each $0 \le j \le \gamma$, check if $R_e > 2^{-j}$. If so, for each $1 \le i \le t$, replace the column of A_i^j that corresponds to element e by the 0 vector.
- Compute $A_i^j w$ for each pair of i and j, and recover a sampled element e_i^j and a weight $w'_{e_i^j}$, which is a $(1+\varepsilon)$-factor approximation to $\left|w_{e_i^j}\right|$.
- For each recovered pair $e_i^j, w'_{e_i^j}$, check if $R_{e_i^j} \le w'_{e_i^j} p$. If there exists such an element e_i^j, output an arbitrary one of them, otherwise output `NoElement`.

Figure 4.3: Weighted edge sampler.

by Proposition 2.3.14 with failure probability $\delta_1, \delta_2 = N^{-100}$ that work for w^j where w^j is the vector generated by sampling each entry of w with probability 2^{-j} (i.e. q in the overview above). For each sampled element e, we check if $w_e p$ is indeed larger than 2^{-j}, if so, e indeed gets sampled and we can output e, otherwise e might not get sampled, and we discard e. We prove that whenever there are elements that get sampled, we will find one of them with high probability.

The analysis is conditioned on the event that none of the γ_1-samplers fails, which is a high probability event. Since the algorithm only uses γ_1 samplers, we can without loss of generality assume that each element has positive weight. Note that the algorithm uses $t\gamma$ number of γ_1 sampling matrices in total, so the total number of measurements of this sketch is $\mathrm{polylog}(N, U, \frac{1}{p})$. Moreover, by Proposition 2.3.14, if the algorithm outputs an element e, we have $R_e \le w'_e \le (1+\varepsilon) w_e p$. it is sufficient to prove that if there exists e such that $R_e \le w_e p$, with high probability, we will not output `NoElement`.

For any $1 \le k \le \gamma$, let S_k be the set of elements e such that $2^{-k} \le w_e p < 2^{-k+1}$, and let S_0 be the set of elements e such that $w_e p \ge 1$. For any $0 \le j \le \gamma$, let S_k^j be the set of elements $e \in S_k$ such that $R_e \le 2^{-j}$. For any j and k, let $n_k = |S_k|$, $n_k^j = \left|S_k^j\right|$ and W_k^j be the total weight of elements in S_k^j. The following claim follows from Chernoff bound.

Claim 4.3.6. *With high probability, for any $0 \le j, k \le \gamma$, if $n_k > 1000 \cdot 2^j \log N$, then $\left|2^j n_k^j - n_k\right| <$*

$n_k/2$, *otherwise* $n_k^j < 1500 \log N$.

Proof. For each element $e \in n_k$, the probability that $e \in n_k^j = 2^{-j}$, so the size of n_k^j is the sum of $|n_k|$ independent random 0/1 variables each with expectation 2^{-j}. The expected size of n_k^j is $2^{-j} n_k$. If $|n_k| > 1000 \cdot 2^j \log N$, by Chernoff bound, the probability that $\left|2^j n_k^j - n_k\right| > n_k/2$ is at most $< 2e^{\frac{n_k}{2^{j+4}}} < N^{-50}$.

If $|n_k| \leq 1000 \cdot 2^j \log N$, the probability that $\left|n_k^j\right| \geq 1500 \log N$ is at most the probability of the case when $n_k = 1000 \cdot 2^j \log N$. So the probability is less than N^{-50}. □

Let k^* be the index that maximizes $\frac{n_{k^*}}{2^{k^*}}$. If $n_{k^*} > 1000 \cdot 2^{k^*} \log N$, by Claim 4.3.6, $n_{k^*}^{k^*} > \frac{n_{k^*}}{2^{k^*+1}}$. Since each element in S_{k^*} has weight at least $2^{-k^*}/p$, $W_{k^*}^{k^*} > \frac{n_{k^*}}{2^{2k^*+1}p}$. On the other hand, for any $k > k^*$, we have $n_k^{k^*} < \max\{1500 \log N, \frac{3n_k}{2^{k^*+1}}\}$ by Claim 4.3.6. Since each element in S_k has weight at most $2^{-k+1}/p$, we have $W_k^{k^*} < \max\{\frac{3000 \log N}{2^k p}, \frac{3n_k}{2^{k+k^*+1}}\}$. As $n_{k^*} > 1000 \cdot 2^{k^*} \log N$, $\frac{3000 \log N}{2^k p} < \frac{3n_{k^*}}{2^{k+k^*}p} < 3W_{k^*}^{k^*}$. Also, by definition of k^*, $\frac{n_{k^*}}{2^{k^*}} \geq \frac{n_k}{2^k}$, which means $\frac{3n_k}{2^{k+k^*+1}} \leq \frac{3n_{k^*}}{2^{2k^*+1}} < 3W_{k^*}^{k^*}$. Thus, we have $W_k^{k^*} < 3W_{k^*}^{k^*}$. Note that for any $1 \leq i \leq t$, $e_i^{k^*}$ is obtained by an γ_1 sampler from $\cup_{k=0}^{\gamma} S_k^{k^*}$, so with probability at least $\frac{1}{3\gamma+3}$, $e_i^{k^*}$ is an element in S_k such that $k \leq k^*$. In this case, $R_e \leq 2^{-k^*} \leq w_e p$, and so the algorithm will output an element that gets sampled. Since $t = 10^4 \gamma \log N$, there exists such an i with high probability.

If $n_{k^*} \leq 1000 \cdot 2^{k^*} \log N$, then for any k, $\frac{n_k}{2^k} \leq 1000 \log N$. Let e be a maximum weight element such that $R_e < w_e p$. Suppose $e \in S_k$, then we have $R_e < w_e p < 2^{-k+1}$. Let k' be the largest index such that $e \in S_k^{k'}$, we have $k' \geq k-1$. For any e' such that $e' \in S_{k''}$ with $k'' < k$, by definition of e and k, $R_{e'} > w_{e'} p \geq 2^{-k''} \geq 2^{-k+1} \geq 2^{k'}$, which means $e' \notin S_{k''}^{k'}$. So $S_{k''}^k = \emptyset$ for any $k'' < k$. On the other hand, for any $k'' \geq k$, $n_{k''}^{k'} < \max\{1500 \log N, \frac{3n_{k''}}{2^{k'}+1}\}$ by Claim 4.3.6. Since any element in $S_{k''}$ has weight at most $2^{-k''+1}/p$, $W_{k''}^{k'} < \{\frac{3000 \log N}{2^{k''}p}, \frac{3n_{k''}}{2^{k''+k'+1}}\}$. Since $w_e \geq \frac{1}{2^k p}$ and $k'' \geq k$, $\frac{3000 \log N}{2^{k''}p} \leq 3000 \log N w_e$. Moreover, since $\frac{n_{k''}}{2^{k''}} \leq 1000 \log N$, $\frac{3n_{k''}}{2^{k''+k'+1}} \leq \frac{3000 \log N}{2^{k'+1}p} \leq 3000 \log N w_e$. So $W_{k''}^{k'} < 3000 \log N w_e$ for any $k'' \geq k$, which means for any $1 \leq i \leq t$, $e_i^{k'} = e$ with probability at least $\frac{1}{3000(\gamma+1)\log N}$. Since $t = 10^4 \gamma \log N$, with high probability, there exists one i such that $e_i^{k'} = e$ and the algorithm will output an element that gets sampled.

So in both cases, the algorithm succeeds with high probability. □

4.4. A linear sketching algorithm for weighted spectral sparsification

In this section, we present a linear sketch with $\tilde{O}(n^{6/5}\varepsilon^{-4})$ measurements that computes a $(1+\varepsilon)$-spectral sparsifier of a weighted graph. Our linear sketch will be an incidence sketch.

This section is structured as follows. First in Section 4.4.1 we prove a key vertex sampling lemma, which says that a heavy edge in G is likely heavy in a random vertex-induced subgraph of G^{sq}. Then in light of this lemma, in Section 4.4.2 we present our linear sketch for recovering heavy edges in G, where we also assume black-box access to two other linear sketches for sparsifying and recovering heavy edges in G^{sq}, respectively. Next in Section 4.4.3, we present our main linear sketching algorithm for weighted spectral sparsification using the heavy edge recovery sketch in Section 4.4.2. Finally in Section 4.4.4 we present the linear sketches for sparsifying and recovering heavy edges in G^{sq} that we invoke in Section 4.4.2.

4.4.1. A vertex sampling lemma

In this subsection we prove a key vertex sampling lemma. This lemma will enable us to recover heavy edges in G by subsampling vertices at rate $\approx n^{-1/5}$ and then recovering edges in the vertex-induced subgraph of G^{sq}.

Lemma 4.4.1 (Vertex sampling lemma)**.** *Let $\eta \in (0,1)$. Given a weighted graph G, let C be a vertex set obtained by including each vertex in G with probability $\frac{\eta}{100n^{1/5}}$ independently. For any edge e in G with leverage score $w_e b_e^T L_G^\dagger b_e \geqslant \eta$, conditioned on $e \in G[C]$, with probability at least .1, its leverage score in $G^{\text{sq}}[C]$ satisfies $w_e^2 b_e^T L_{G^{\text{sq}}[C]}^\dagger b_e \geqslant 1/1000$.*

Roughly, our proof of the lemma proceeds as follows: (i) group vertices according to their potentials induced by an electrical flow between the endpoints of e in G; (ii) analyze the structure of the edges in the vertex-induced subgraph based on their weights and the potential difference between their endpoints; (iii) explicitly construct a set of vertex potentials in G^{sq} that certifies the heaviness of the edge e.

Proof of Lemma 4.4.1. Let $e = (s,t)$ and without loss of generality, assume $w_e = 1$ (since we could always scale all edge weights simultaneously without changing any leverage scores). Since the leverage score $\tau_e \geqslant \eta$, we also have that the effective resistance $b_e^T L_G^\dagger b_e \geqslant \eta$. We use the electrical

network view of the graph G, and let $x = \frac{L_G^\dagger b_e}{b_e^T L_G^\dagger b_e} \in \mathbb{R}^n$ be the set of vertex potentials induced by an electrical flow from s to t of $\frac{1}{b_e^T L_G^\dagger b_e} \leqslant \frac{1}{\eta}$ units. We also assume without loss of generality $x_t = 0$ (since we could always shift all vertex potentials by x_t otherwise). Since $b_e^T L_G^\dagger b_e$ is the effective resistance between s and t, we have $x_s = 1$ by Ohm's law. Moreover, by Fact 2.4.5, the normalized energy of x w.r.t. s, t satisfies that

$$x^T L_G x = \frac{1}{b_e^T L_G^\dagger b_e} \leqslant 1/\eta.$$

We now partition the vertices other than s and t into $n^{4/5}$ groups based on their potentials. Specifically, the i^{th} group S_i contains all vertices satisfying $(i-1)\cdot n^{-4/5} \leqslant x_u \leqslant i\cdot n^{-4/5}$, where we break ties arbitrarily. For an edge $f = (u, v)$, we say f passes through S_i if $x_u \leqslant (i-1)\cdot n^{-4/5} < i\cdot n^{-4/5} \leqslant x_v$ or $x_v \leqslant (i-1)\cdot n^{-4/5} < i\cdot n^{-4/5} \leqslant x_u$.

Claim 4.4.2. *For any $i \in [n^{4/5}]$, the total weight of edges that pass through S_i is at most $n^{4/5}/\eta$.*

Proof. Consider an edge $f = (u, v)$ that passes through S_i, and assume without loss of generality $x_u \geqslant x_v$. By Ohm's law, the flow on edge f is in the direction $u \to v$ and has amount $w_f(x_u - x_v) \geqslant w_f n^{-4/5}$. Since the total amount of flow across the cut

$$(\{t\} \cup S_1 \cup \ldots \cup S_i,\ S_{i+1} \cup \ldots \cup S_{n^{4/5}} \cup \{s\})$$

is at most $1/\eta$, we have

$$\sum_{f:\ f \text{ passes through } S_i} w_f n^{-4/5} \leqslant 1/\eta,$$

which means that the total weight of such edges is at most $n^{4/5}/\eta$. □

We now consider what happens when we look at a vertex-induced subgraph $G[C]$ where C is obtained by including each vertex (other than s, t) with probability $\frac{\eta}{100n^{1/5}}$, and then also including s, t. We say an edge $f = (u, v)$ is *intermediate* if $\{u, v\} \cap \{s, t\} = \emptyset$. We say a group S_i is *good* if (i) none of the vertices in S_i gets sampled in C, and (ii) all intermediate edges that pass through S_i and have

both endpoints in C have weight at most $n^{2/5}$. We say S_i is *bad* otherwise.

Claim 4.4.3. *With probability at least 2/3, the number of good S_i's with $i \in (\frac{1}{4}n^{4/5}, \frac{3}{4}n^{4/5}]$ is at least $n^{4/5}/20$.*

Proof. First, by Markov's inequality, at least .8 fraction of the S_i's with $i \in (\frac{1}{4}n^{4/5}, \frac{3}{4}n^{4/5}]$ have size at most $10n^{1/5}$. For any fixed S_i with $|S_i| \leqslant 10n^{1/5}$, we have

$$\mathbb{E}\left[|S_i \cap C|\right] \leqslant \frac{|S_i|\eta}{100n^{1/5}} \leqslant \frac{\eta}{10} \leqslant \frac{1}{10}.$$

Therefore, once again by Markov's inequality, the probability that (i) happens for any fixed S_i with $|S_i| \leqslant 10n^{1/5}$ is at least .9.

On the other hand, by Claim 4.4.2, the total number of edges with weight $> n^{2/5}$ that pass through S_i is at most $n^{2/5}/\eta$. The probability of any intermediate edge belonging to $G[S]$ is $\left(\frac{\eta}{100n^{1/5}}\right)^2 = \frac{\eta^2}{10000n^{2/5}}$. These combined give us that the expected number of intermediate edges with weight $> n^{2/5}$ that pass through S_i and have both endpoints in C is at most $1/10000$. Now an application of Markov's inequality gives that (ii) happens for S_i with probability at least $1 - 1/10000$.

Therefore, by a union bound, each S_i with $|S_i| \leqslant 10n^{1/5}$ is good with probability at least .89. Thus the expected number of bad S_i's with $|S_i| \leqslant 10n^{1/5}$ is at most $.11n^{4/5}$. By Markov's inequality, the number of bad S_i's with $|S_i| \leqslant 10n^{1/5}$ is at most $.33n^{4/5}$ with probability $\geqslant 2/3$. Thus, with probability at least 2/3, the number of good S_i's with $i \in (\frac{1}{4}n^{4/5}, \frac{3}{4}n^{4/5}]$ is at least $.4n^{4/5} - .33n^{4/5} \geqslant .05n^{4/5}$, as desired. □

We will now construct a set of vertex potentials (call it $y \in \mathbb{R}^{|C|}$) in $G[C]$ from x. We will show that the energy of the new set of potentials is small in $G^{\text{sq}}[C]$ (i.e. even with edge weights squared), but the potential difference between s, t is still large, which result in a small normalized energy w.r.t. s, t, and thus certify the "heaviness" of edge (s, t) in $G^{\text{sq}}[C]$.

Specifically, we obtain y by "collapsing" the vertex potentials within each bad S_i, so that the intermediate edges that do not pass through any good S_i's will have both endpoints getting the same potential. To take care of the edges incident on s or t that span less than 1/4 fraction of the

groups, we will also collapse the vertex potentials in the range $[0, 1/4]$ and $[3/4, 1]$. Precisely, y_u is given as follows for each $u \in C$:

1. If $x_u \geqslant 3/4$, then set $y_u \leftarrow 1$.
2. Otherwise, suppose $x_u \in S_i$ for some $i \leqslant \frac{3}{4}n^{4/5}$. Count the number of good S_j's with $j \in (\max\left\{i, \frac{1}{4}n^{4/5}\right\}, \frac{3}{4}n^{4/5}]$ and let k denote that number. Then set $y_u \leftarrow 1 - kn^{-4/5}$.

Claim 4.4.4. *y satisfies the following properties:*

1. *For any u, v, $|y_u - y_v| \leqslant |x_u - x_v|$.*
2. *With probability at least $2/3$, $y_s - y_t \geqslant 1/20$.*
3. *With probability at least $.8$, $y^T L_{G^{\mathrm{sq}}[C]} y \leqslant 2$.*

Proof. Note that for any i, $C \cap S_i \neq \emptyset$ implies that S_i is bad. Therefore, by our construction, all vertices within the same bad S_i will end up having the same potentials in y. As a result, for any u, v, $|y_u - y_v|$ equals $n^{-4/5}$ times the number of good groups S_i between x_u and x_v such that $i \in (\frac{1}{4}n^{4/5}, \frac{3}{4}n^{4/5}]$, which implies 1.

By Claim 4.4.3, with probability $2/3$, the number of good S_i's with $i \in (\frac{1}{4}n^{4/5}, \frac{3}{4}n^{4/5}]$ is at least $n^{4/5}/20$. Thus we have 2.

We then prove 3. First note that for edges that do not pass through any good S_i, both their endpoints have the same potential in y. So the total energy contributed by these edges is zero. Now the remaining edges can be divided into two types: (A) edges that are incident on s or t; (B) intermediate edges that pass through some good S_i. By the definition of good S_i's, edges of type (B) have weight at most $n^{2/5}$ each.

For edges of type (A), they are of the form (s, u) or (v, t). If $x_u \geqslant 3/4$ or $x_v \leqslant 1/4$, then once again both endpoints of the edge have the same potential in y, and the energy contribution from such edges is zero. We thus focus on the edges of type (A) such that $x_u < 3/4$ or $x_v > 1/4$, and refer to those edges as type (A'). For any type (A') edge $f = (a, b)$, we have $|x_a - x_b| \geqslant 1/4$, and thus by Ohm's law the amount of flow on f is at least $w_f/4$. Since the total amount of flow going out of s or going into t is upper bounded by $1/\eta$, the total weight of type (A') edges is at most $8/\eta$. This

means that the number of type (A') edges f with $w_f > 1$ is at most $8/\eta$. Thus the expected number of such edges in $G[C]$ is at most $(8/\eta)\frac{\eta}{100n^{1/5}} \leqslant \frac{1}{10}$. By Markov's inequality, none of such edges is in $G[C]$ with probability at least .9 (call this event $\mathcal{E}_1$). As for any type (A') edge $f = (a, b)$ with $w_f \leqslant 1$, we have $w_f^2(y_a - y_b)^2 \leqslant w_f(x_a - x_b)^2$. This combined with the fact that each f belongs to $G[C]$ with probability $\frac{\eta}{100n^{1/5}}$, the expected contribution of type (A') edges with weight at most 1 to $y^T L_{G^{\mathrm{sq}}[C]} y$ is at most $\frac{\eta}{100n^{1/5}} \cdot x^T L_G x \leqslant \frac{1}{100n^{1/5}} \leqslant \frac{1}{100}$. Thus this contribution does not exceed 1 with probability .99 (call this event $\mathcal{E}_2$).

Finally we consider type (B) edges. Since their weights are at most $n^{2/5}$ each, squaring the edge weights blows up the energy on them by at most a factor of $n^{2/5}$. On the other hand, since these are intermediate edges, the probability that any such edge belongs to $G[C]$ is $\left(\frac{\eta}{100n^{1/5}}\right)^2 = \frac{\eta^2}{10000n^{2/5}}$. Therefore, the expected total contribution of type (B) edges to $y^T L_{G^{\mathrm{sq}}[C]} y$ is at most $\frac{\eta^2}{10000n^{2/5}} \cdot n^{2/5} x^T L_G x \leqslant \frac{\eta}{10000} \leqslant \frac{1}{10000}$. Thus this contribution does not exceed 1 with probability $1 - 1/10000$ (call this event $\mathcal{E}_3$).

By a union bound, $\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3$ simultaneously happen with probability at least .8, in which case we have $y^T L_{G^{\mathrm{sq}}[C]} y \leqslant 2$. □

By Claim 4.4.4 and a union bound over the events in the claim, we have with probability at least .1 that the normalized energy of y with respect to s, t in $G^{\mathrm{sq}}[C]$ is $\frac{y^T L_{G^{\mathrm{sq}}[C]} y}{(y_s - y_t)^2} \leqslant 800$. □

4.4.2. Recovery of heavy edges

Armed with the vertex sampling lemma, we are now ready to design a linear sketch to recover all heavy edges in G. In doing so, we will also need to invoke two other linear sketches for sparsifying and recovering heavy edges in G^{sq}, respectively. We summarize their performance in the two lemmas below, and prove them later in Section 4.4.4. We note that the sketch designed in Lemma 4.4.6 is basically a direct application of ℓ_2-heavy hitters. The sketch designed in Lemma 4.4.5 is essentially a reduction from sparsification to heavy edge recovery, which will be very similar to our main linear sketching algorithm for weighted spectral sparsification in the next subsection (Section 4.4.3).

Lemma 4.4.5. *For any parameter $\varepsilon_2 > 0$ and any integer n, there exists a linear sketch with sketching matrix $S^{\mathrm{sq}} \in \mathbb{R}^{n\varepsilon_2^{-4}\,\mathrm{polylog}(n,\varepsilon_2^{-1},\frac{w_{\max}}{w_{\min}}) \times \binom{n}{2}}$ and recovery algorithm* `square-recovery` *such*

that, given an input graph G of n vertices with weight vector w_G, `square-recovery`$(S^{\text{sq}}w_G)$ *returns a $(1+\varepsilon_2)$-spectral sparsifier of G^{sq} with high probability.*

Lemma 4.4.6. *For any $\eta_3, \varepsilon_3 \in (0,1)$ and any integer n, there exists a linear sketch with sketching matrix $S^{\text{sqh}} \in \mathbb{R}^{n\eta_3^{-1}\varepsilon_3^{-2}\operatorname{polylog}(n)\times\binom{n}{2}}$ and recovery algorithm* `square-heavy-recovery` *such that, for an input graph G of n vertices with weight vector w_G and another graph $\tilde{G}$, the subroutine* `square-heavy-recovery`$(S^{\text{sqh}}w_G, \tilde{G})$ *recovers a set of edges F in G along with estimates of their weights $\tilde{w}_f$'s such that with high probability*

1. *F contains all edges e satisfying*

$$\frac{((w_G)_e b_e^T L_{\tilde{G}}^{\dagger} b_e)^2}{b_e^T L_{\tilde{G}}^{\dagger} L_{G^{\text{sq}}} L_{\tilde{G}}^{\dagger} b_e} \geqslant \eta_3.$$

2. *All edges $f \in F$ satisfy $\frac{1}{1+\varepsilon_3}(w_G)_f \leqslant \tilde{w}_f \leqslant (1+\varepsilon_3)(w_G)_f$.*

We remark that to understand the first guarantee of the above lemma, one should think of $\tilde{G}$ as a good spectral sparsifier of G^{sq}, so that the numerator $((w_G)_e b_e^T L_{\tilde{G}}^{\dagger} b_e)^2 \approx ((w_G)_e b_e^T L_{G^{\text{sq}}}^{\dagger} b_e)^2$ and the denominator $b_e^T L_{\tilde{G}}^{\dagger} L_{G^{\text{sq}}} L_{\tilde{G}}^{\dagger} b_e \approx b_e^T L_{G^{\text{sq}}}^{\dagger} b_e$, and thus the LHS $\approx (w_G)_e^2 b_e^T L_{G^{\text{sq}}}^{\dagger} b_e$, the leverage score of e in G^{sq}. That is, this guarantee is essentially saying that all heavy edges in G^{sq} will be recovered.

We now describe the linear sketch for recovering heavy edges in G in Figure 4.4, and characterize its performance in the lemma below.

Lemma 4.4.7. *For any parameters $\eta_1 \in (n^{-4/5}\log n, 1)$, $\varepsilon_1 \in (0,1)$ and integer n, there exists a linear sketch with sketching matrix $S^{\text{hv}} \in \mathbb{R}^{n^{6/5}\eta_1^{-1}\varepsilon_1^{-2}\operatorname{polylog}(n, \frac{w_{\max}}{w_{\min}})\times\binom{n}{2}}$ and recovery algorithm* `heavy-edge-recovery` *such that, for an input graph G of n vertices with weight vector w_G,* `heavy-edge-recovery`$(S^{\text{hv}}w_G)$ *recovers a set F of edges in G along with estimates of their weights $\tilde{w}_f$'s such that with high probability*

1. *All edges e whose leverage score in G satisfy $(w_G)_e b_e^T L_G^{\dagger} b_e \geqslant \eta_1$ belong to F.*
2. *All edges $f \in F$ satisfy $\frac{1}{1+\varepsilon_1}(w_G)_f \leqslant \tilde{w}_f \leqslant (1+\varepsilon_1)(w_G)_f$.*

We now prove Lemma 4.4.7 using Lemmas 4.4.1,4.4.5,4.4.6.

Proof of Lemma 4.4.7 using Lemmas 4.4.1,4.4.5,4.4.6. **Number of linear measurements.** We

`heavy-edge-sketch`$(G, \eta_1, \varepsilon_1)$

1. Let $t = \left\lceil 10000\eta_1^{-2} n^{2/5} \log n \right\rceil$ and let $V_1, \ldots, V_t$ be vertex subsets, each obtained by including each vertex in G with probability $\frac{\eta_1}{100n^{1/5}}$ independently.
   (Subsample the vertices sufficiently many times to cover every edge)
2. For each $i \in [t]$, let $S_i^{\text{sq}} \in \mathbb{R}^{|V_i|\varepsilon_2^{-4}\,\text{polylog}(|V_i|,\varepsilon_2^{-1},\frac{w_{\max}}{w_{\min}})\times\binom{|V_i|}{2}}$ be a sketching matrix with error $\varepsilon_2 = .01$ (Lemma 4.4.5), and let $S_i^{\text{sqh}} \in \mathbb{R}^{|V_i|\eta_3^{-1}\varepsilon_3^{-2}\,\text{polylog}(|V_i|)\times\binom{|V_i|}{2}}$ with $\eta_3 = 1/2000$ and $\varepsilon_3 = \varepsilon_1/100$ (Lemma 4.4.6).
3. Concatenate the following sketches as $S^{\text{hv}} w_G$:
   (a) $S_1^{\text{sq}} w_{G[V_1]}, \ldots, S_t^{\text{sq}} w_{G[V_t]}$, where $w_{G[V_i]} \in \mathbb{R}^{\binom{|V_i|}{2}}$ is the weight vector of the vertex induced subgraph $G[V_i]$.
   (Create a sparsification sketch for each vertex-induced subgraph)
   (b) $S_1^{\text{sqh}} w_{G[V_1]}, \ldots, S_t^{\text{sqh}} w_{G[V_t]}$.
   (Create a heavy edge sketch for each vertex-induced subgraph)

`heavy-edge-recovery`$(S^{\text{hv}} w_G)$

1. For each $i \in [t]$:
   (a) Use Lemma 4.4.5 to recover from $S_i^{\text{sq}} w_{G[V_i]}$ a sparsifier $\tilde{G}_i$ of $G^{\text{sq}}[V_i]$.
   ($\tilde{G}_i$ is a 1.01-spectral sparsifier of $G^{\text{sq}}[V_i]$)
   (b) Use the recovery algorithm from Lemma 4.4.6 to recover from $S_i^{\text{sqh}} w_G[V_i]$ and $\tilde{G}_i$ a set F of edges along with their estimated weights $\tilde{w}_f$'s, and mark all edges in F as heavy.
   (Feed the sparsifier $\tilde{G}_i$ to the recovery algorithm to get all heavy edges)
2. Return all edges marked heavy along with the estimates of their weights (if for some edge there are multiple estimates of its weight, pick an arbitrary one).

Figure 4.4: Linear sketch for recovering heavy edges in G.

observe that with high probability, $|V_i| \leqslant 100\eta_1 n^{4/5}$ for all $i \in [t]$, and thus each $S_i^{\mathrm{sq}} w_G[V_i] \in \mathbb{R}^{\eta_1 n^{4/5} \operatorname{polylog}(n, \frac{w_{\max}}{w_{\min}})}$ and each $S_i^{\mathrm{sqh}} w_G[V_i] \in \mathbb{R}^{\eta_1 n^{4/5} \varepsilon_1^{-2} \operatorname{polylog}(n)}$. Therefore, the total number of linear measurements is bounded by

$$t \cdot \eta_1 n^{4/5} \varepsilon_1^{-2} \operatorname{polylog}(n, \frac{w_{\max}}{w_{\min}}) \leqslant n^{6/5} \eta_1^{-1} \varepsilon_1^{-2} \operatorname{polylog}(n, \frac{w_{\max}}{w_{\min}}).$$

Guarantee 1. Consider fixing any edge e with $w_e b_e^T L_G^\dagger b_e \geqslant \eta_1$. By Lemma 4.4.1, for each $i \in [t]$, with probability $\frac{\eta_1^2}{1000 n^{2/5}}$, we have $e \in G[V_i]$ and $w_e^2 b_e^T L_{G^{\mathrm{sq}}[V_i]}^\dagger b_e \geqslant 1/1000$ (call this event $\mathcal{E}_i$). Therefore at least one of $\mathcal{E}_1, \ldots, \mathcal{E}_t$ happens with probability $\geqslant 1 - (1 - \frac{\eta_1^2}{1000 n^{2/5}})^t \geqslant 1 - 1/n^{-10}$. Whenever an $\mathcal{E}_i$ happens, using the fact that $\tilde{G}_i$ is a 1.01-spectral sparsifier of $G^{\mathrm{sq}}[V_i]$ (by Lemma 4.4.5), we have

$$(w_G)_e b_e^T L_{\tilde{G}_i}^\dagger b_e \geqslant \frac{1}{1.01} (w_G)_e b_e^T L_{G^{\mathrm{sq}}[V_i]}^\dagger b_e$$

and

$$b_e^T L_{\tilde{G}_i}^\dagger L_{G^{\mathrm{sq}}[V_i]} L_{\tilde{G}_i}^\dagger b_e \leqslant 1.01 b_e^T L_{\tilde{G}_i}^\dagger L_{\tilde{G}_i} L_{\tilde{G}_i}^\dagger b_e = b_e^T L_{\tilde{G}_i}^\dagger b_e \leqslant 1.01^2 b_e^T L_{G^{\mathrm{sq}}[V_i]}^\dagger b_e,$$

and thus

$$\frac{((w_G)_e b_e^T L_{\tilde{G}_i}^\dagger b_e)^2}{b_e^T L_{\tilde{G}_i}^\dagger L_{G^{\mathrm{sq}}[V_i]} L_{\tilde{G}_i}^\dagger b_e} \geqslant 1.01^{-3} (w_G)_e^2 b_e^T L_{G^{\mathrm{sq}}[V_i]}^\dagger b_e \geqslant 1.01^{-3}/1000 \geqslant 1/2000.$$

By Lemmas 4.4.6, with high probability, e is among the recovered edges. Therefore by a union bound over all such edges e, we have the desired result.

Guarantee 2. This follows directly from Lemma 4.4.6. □

4.4.3. Main algorithm for weighted spectral sparsification

We now show how to use the heavy edge recovery sketch in the previous section to obtain a spectral sparsifier of G.

We first briefly summarize the main ideas. The first idea is to use the iterative refinement process

in [LMP13] as in the previous works on unweighted graphs [KLM+14, KMM+20]. That is, we consider, for some large $\alpha = \text{poly}(w_{\max}, n)$, $t = \text{polylog}(\varepsilon^{-1}, \frac{w_{\max}}{w_{\min}}, n)$ and a constant $\beta \in (0, 1)$, the sequence of graphs

$$G + \alpha K_n, G + \beta\alpha K_n, G + \beta^2\alpha K_n, \ldots, G + \beta^t \alpha K_n,$$

which have the properties that

1. αK_n is an $O(1)$-spectral sparsifier of $G + \alpha K_n$.
2. $G + \beta^k \alpha K_n$ is an $O(1)$-spectral sparsifier of $G + \beta^{k+1}\alpha K_n$ for all $k \geqslant 0$.
3. $G + \beta^t \alpha K_n$ is a $(1 + \varepsilon)$-spectral sparsifier of G.

The idea is then to iteratively obtain a sparsifier of each of these graphs, where we use the sparsifier of $G + \beta^k \alpha K_n$ to guide the sparsification of $G + \beta^{k+1}\alpha K_n$ (in particular, we use the sparsifier of the former to estimate the effective resistances and leverage scores in the latter). Thus it boils down to how to sparsify $G^{(k+1)} := G + \beta^{k+1}\alpha K_n$ using heavy edge recovery.

Remark 4.4.8. *We remark that, when given access to a heavy edge recovery sketch, the sparsification of $G^{(k+1)}$ is relatively easy to achieve in the unweighted case, for the following reason. Consider, in an unweighted graph, an edge $e = (s, t)$ with effective resistance (thus also leverage score) r_e, and let $x_{s,t} \in \mathbb{R}^n$ be the set of vertex potentials induced by an electrical flow from s to t and assume w.l.o.g. $x_s - x_t = 1$. By Fact 2.4.5, we have $x^T L_G x = 1/r_e$. Now notice that we can also assume $x_u \in [0, 1]$ for all u, since letting $x_u \leftarrow 1$ for all $x_u > 1$ and $x_v \leftarrow 0$ for all $x_v < 0$ can only decrease the total energy.*

This means that the energy $(x_s - x_t)^2 = 1$ on edge $e = (s, t)$ is the largest among all edges, and thus if we sample all edges uniformly at rate $\approx r_e$, the total energy will be $\widetilde{O}(1)$ with high probability by Chernoff bounds. This implies that in the latter subsampled graph, e (if sampled) becomes heavy with high probability. Since r_e is exactly (up to an $O(\log n)$ factor) the probability with which we want to sample e, we can apply the heavy edge sketch to subgraphs of G obtained by sampling edges at geometrically decreasing rates, and then try to recover each edge e from the subgraph with sampling rate $\approx r_e$.

However, for weighted graphs, the energy on some other edges of very large weights can be unboundedly big. Thus concentration bounds no longer give us high success probability of recovering an edge e when sampling all edges uniformly at rate $\approx \tau_e = w_e r_e$, even though the energy reduces significantly in expectation.

To sparsify $G^{(k+1)}$, we will utilize the spectral sparsification framework in [Kou14], which is itself model oblivious. The framework works as follows:

1. Fix some constant $p \in (0, 1)$.
2. While the number of edges in the graph is $> n\varepsilon^{-2}\operatorname{polylog}(n)$:
   (a) Find all edges whose leverage score $\geqslant \eta := \varepsilon^2/\operatorname{polylog}(n)$, and call these edges F.
   (b) Sample each edge not in F with probability p, and multiply its weight by $1/p$ if sampled.

Notice that since the leverage scores of all edges sum up to at most $n - 1$, the total number of edges in F is at most $n\varepsilon^{-2}\operatorname{polylog}(n)$. Thus, in each while loop iteration, the number of edges decreases by a constant factor, and as a result there can be at most $O(\log n)$ iterations. Then using Theorem 2.4.6, we have that the final graph is a $(1 + \varepsilon)$-spectral sparsifier of $G^{(k+1)}$. Notice that here, the first step in the while loop is exactly the recovery of heavy edges.

We now describe the difficulty that arises in implementing the above process using linear sketching *non-adaptively*, and our way around it. First, let $E_0 \supseteq E_1 \supseteq \ldots \supseteq E_{O(\log n)}$ be such that $E_0 = \binom{V}{2}$ and E_{i+1} is obtained by subsampling each edge slot in E_i with probability p (the constant fixed at the first step of the above process). We apply the heavy edge recovery sketch to each $G^{(k+1)}[E_i]$. We then implement each iteration of the while loop in the above process.

At first, we recover all heavy edges in $G^{(k+1)}$ using the sketch of $G^{(k+1)}[E_0] = G^{(k+1)}$, and call these edges F_0. We would like to sample each edge in $G^{(k+1)}[\binom{V}{2} \setminus F_0]$ with probability p, and multiply its weight by $1/p$ if sampled. Then in the next iteration, we would want to recover heavy edges in the latter subsampled graph. That is, we would like to have a sketch of the graph $(1/p)G^{(k+1)}[E_1 \setminus F_0] + G^{(k+1)}[F_0]$. However, we only have a sketch of $G^{(k+1)}[E_1]$. By linearity, we can multiply it by $1/p$ and add to it $G^{(k+1)}[F_0 \setminus E_1]$ and get a sketch of $(1/p)G^{(k+1)}[E_1] + G^{(k+1)}[F_0 \setminus E_1]$. Nonetheless, this sketch is still not taken on our desired graph $(1/p)G^{(k+1)}[E_1 \setminus F_0] + G^{(k+1)}[F_0]$,

since the weights of the edges in $E_1 \cap F_0$ in the former graph are larger than in the latter by a factor of $1/p$. We say these edges are *overweighted* by $1/p$, and call the former graph *overweighted* graph.

One might hope to further subtract from the sketch $(1/p - 1)G^{(k+1)}[E_1 \cap F_0]$ to bring down the weights of the overweighted edges by a factor of $(1/p)$. However, notice that we do *not* have the exact weights of the edges in F_0 from our heavy edge recovery sketch. Rather, we only have some estimates of their weights. Moreover, while in the second iteration we are only looking to subtract edges that are overweighted by $1/p$, in subsequent iterations, we might need to subtract edges that are overweighted by $\text{poly}(n)$, which means that our weight estimates for such edges must have inverse polynomial accuracy for the subtraction to work.

Our way around this issue is to repeatedly *re-estimate* the weights of the overweighted edges. Specifically, we show that the edges that are overweighted the most must be heavy in the overweighted graph. Thus we can apply the heavy edge recovery sketch to the overweighted graph, get estimates of the weights of these edges, and bring their weights down by a factor of $(1/p)$. We then repeatedly apply this step $O(\log n)$ times (where we re-estimate the weights each time) until there are no overweighted edges. Since we only bring down the edge weights by a constant factor each time and always re-estimate the weights once changed, we will never have too large an error.

We now present in Figure 4.5 our main algorithm for weighted spectral sparsification, which invokes the heavy edge recovery sketch in Section 4.4.2. Specifically, for each graph $G^{(k)}$ in the iterative refinement process, we apply independent heavy edge recovery sketches to subgraphs of $G^{(k)}$ obtained by sampling edges at geometrically decreasing rates. Then in the recovery step, we first simulate the iterative refinement process using an outer for loop of k, and then implement the framework from [Kou14] in an inner for loop of i. Inside each iteration of the inner for loop, we start with the sketch of an overweighted graph Z, and then gradually bring down the weights of the overweighted edges by repeatedly recovering heavy edges in the current (overweighted) graph and subtracting a constant fraction of their weights. Finally when there are no overweighted edges left, we recover the heavy edges in the resulting graph and then go to the next sampling rate.

The performance of our main linear sketching algorithm for weighted graph sparsification is characterized in Theorem 4.4.9.

`spectral-sketch`(G, ε)

1. Let $t = \left\lceil 10000 \log(10^6 \varepsilon^{-1} \frac{w_{\max}}{w_{\min}} \cdot n^{10}) \right\rceil$, and then for each $k = 0, 1, 2, \ldots, t$:
   (a) Let $E_0^{(k)} \supseteq E_1^{(k)} \supseteq \ldots \supseteq E_t^{(k)}$ be edge subsets where $E_0^{(k)} = \binom{V}{2}$ and $E_i^{(k)}$ is obtained by sub-sampling each edge slot in $E_{i-1}^{(k)}$ with probability $(1 + 1/1000)^{-1}$.
   (b) For each pair $0 \leqslant i, j \leqslant t$, use Lemma 4.4.7 to generate a sketching matrix $(S^{\text{hv}})_{i,j}^{(k)} \in \mathbb{R}^{O(n^{6/5}\eta_1^{-1}\varepsilon_1^{-2}\,\text{polylog}(n, \frac{w_{\max}}{w_{\min}}))\times\binom{n}{2}}$ with $\eta_1 = \varepsilon^2/(10^{12}t^2 \log n)$, $\varepsilon_1 = \varepsilon/(10^6 t)$.
2. For each $0 \leqslant k, i, j \leqslant t$, let $G^{(k)} \leftarrow G + (1 + 1/10^4)^{-k} 10^6 w_{\max} n^5 K_n$, and compute the sketch $(S^{\text{hv}})_{i,j}^{(k)} w_{G^{(k)}[E_i^{(k)}]}$, where $w_{G^{(k)}[E_i^{(k)}]} \in \mathbb{R}^{\binom{n}{2}}$ is the weight vector of $G^{(k)}[E_i^{(k)}]$.
   (Take sufficiently many independent heavy edge sketches on each subsampled graph)
3. Concatenate these sketches as $S^{\text{ss}} w_G$.

$H^{(t)} =$ `spectral-recovery`$(S^{\text{ss}} w_G)$

1. Initially, let $H^{(0)} \leftarrow 10^6 w_{\max} n^5 K_n$. ($H^{(0)}$ is a 1.001-spectral sparsifier of $G^{(0)}$)
2. For $k = 1, 2, \ldots, t$:
   (a) Let $H^{(k)} \leftarrow \emptyset$. ($H^{(k)}$ will be a $(1 + \varepsilon/1000)$-spectral sparsifier of $G^{(k)}$)
   (b) Set $c_e \leftarrow 0$ for all $e \in \binom{V}{2}$. (c_e will be s.t. e is added to $H^{(k)}$ when $i = c_e$)
   (c) For $i = 0, 1, \ldots, t$:
      i. Let $Z \leftarrow (1 + 1/1000)^i G^{(k)}[E_i^{(k)}] + H^{(k)}[\binom{V}{2} \setminus E_i^{(k)}]$.
         (Z records the graph on which our linear sketches are currently taken)
      ii. Compute sketches $s_j := (S^{\text{hv}})_{i,j}^{(k)} w_Z, j \in [0, t]$, where w_Z is Z's weight vector.
      iii. For each $f \in H^{(k)} \cap E_i^{(k)}$, let $\delta_f \leftarrow i - c_f$, and let $\delta_f \leftarrow 0$ for all other edges.
         ($(w_Z)_f$ needs to be brought down by a factor of $(1 + 1/1000)^{\delta_f}$)
      iv. Let $j \leftarrow 0$. Then while $\exists f : \delta_f > 0$, do the following:
         - Use Lemma 4.4.7 to recover from s_j a set F of edges and then let $j \leftarrow j + 1$.
         - For each $f \in F$ such that $\delta_f > 0$, let $\tilde{w}_f$ be the estimate of its weight:
           - $Z \leftarrow Z - (1 - (1 + 1/1000)^{-1})\tilde{w}_f f$.
           - $s_{j'} \leftarrow s_{j'} - (1 - (1 + 1/1000)^{-1})(S^{\text{hv}})_{i,j'}^{(k)}(\tilde{w}_f \chi_f)$ for all $j' \in [0, t]$.
           - $\delta_f \leftarrow \delta_f - 1$.
             (Bring down $(w_Z)_f$ by $(1 + 1/1000)$ and update all sketches accordingly)
      v. Use Lemma 4.4.7 to recover from s_j a set F^* of edges.
      vi. For each edge $f \in F^*$ with estimated weight $\tilde{w}_f$ such that $\tilde{w}_f b_f^T L_{H^{(k-1)}}^{\dagger} b_f \geqslant 8\eta_1$ and f is not already in $H^{(k)}$, add f to $H^{(k)}$ with weight $\tilde{w}_f$, and let $c_f \leftarrow i$.
         (Add recovered heavy edges to $H^{(k)}$, then go to the next sampling rate)

Figure 4.5: Linear sketch for weighted spectral sparsification.

Theorem 4.4.9. *For any parameter $\varepsilon > 0$ and any integer n, there exists a linear sketch with sketching matrix $S^{\text{ss}} \in \mathbb{R}^{n^{6/5}\varepsilon^{-4}\,\text{polylog}(n,\varepsilon^{-1},\frac{w_{\max}}{w_{\min}})\times\binom{n}{2}}$ and recovery algorithm* `spectral-recovery` *such that, given an input graph G of n vertices with weight vector w_G,* `spectral-recovery`*$(S^{\text{ss}}w_G)$ returns a $(1+\varepsilon)$-spectral sparsifier of G with high probability.*

Before proving the theorem, we shall first give some useful intermediate lemmas. The following proposition directly follows from the definition of spectral sparsifiers.

Proposition 4.4.10. *For any two graphs G_1, G_2 whose weight vectors satisfy that $(w_{G_1})_e \approx_{1+\varepsilon} (w_{G_2})_e$ for all $e \in \binom{V}{2}$, G_1 is a $(1+\varepsilon)$-spectral sparsifier of G_2.*

This proposition then immediately implies the following two lemmas.

Lemma 4.4.11. *$10^6 w_{\max} n^5 K_n$ is a 1.001-spectral sparsifier of $G^{(0)}$.*

Lemma 4.4.12. *For all $k \geqslant 1$, $G^{(k-1)}$ is a 1.001-spectral sparsifier of $G^{(k)}$.*

Lemma 4.4.13. *$G^{(t)}$ is a $(1+\varepsilon/2)$-spectral sparsifier of G.*

Proof. By definition

$$\begin{aligned} L_{G^{(t)}} =& L_G + L_{(1+10^{-4})^{-t}10^6 w_{\max} n^5 K_n} \\ \preceq& L_G + .1\varepsilon w_{\min} n^{-5} L_{K_n}, \end{aligned}$$

where the last line follows from our choice of t. Thus, the largest eigenvalue of the second term is bounded by $.1\varepsilon w_{\min} n^{-4}$. By standard lower bounds on the second smallest eigenvalue, the second smallest eigenvalue of L_G is at least $w_{\min}/n^2$. Therefore we have

$$-.1\varepsilon L_G \preceq L_{G^{(t)}} - L_G \preceq .1\varepsilon L_G,$$

which implies that $G^{(t)}$ is a $(1+\varepsilon/2)$-spectral sparsifier of G. □

Fix an iteration of the outer for loop of k. Then for an iteration of the inner for loop of i, let $H_i^{(k)}$

be the $H^{(k)}$ at the *beginning* of the iteration, and let $F_i^{(k)}$ be the edges in $H_i^{(k)}$. Define graph

$$J_i^{(k)} := (1+1/1000)^i G^{(k)}[E_i^{(k)} \setminus F_i^{(k)}] + \sum_{\ell=0}^{i-1}(1+1/1000)^\ell G^{(k)}[F_{\ell+1}^{(k)} \setminus F_\ell^{(k)}].$$

Notice that $F_0^{(k)} = \emptyset$ and $J_0^{(k)} = G^{(k)}$. Also by the way we are assigning values to c_f in the algorithm, we have, at the beginning of the for loop (of i) iteration, $f \in F_{c_f+1}^{(k)} \setminus F_{c_f}^{(k)}$ for all $f \in F_i^{(k)}$. Thus we also have

$$J_i^{(k)} := (1+1/1000)^i G^{(k)}[E_i^{(k)} \setminus F_i^{(k)}] + \sum_{f \in F_i^{(k)}} (1+1/1000)^{c_f}(w_{G^{(k)}})_f f. \tag{4.9}$$

Lemma 4.4.14. *Suppose $H^{(k-1)}$ is a 1.001-spectral sparsifier of $G^{(k-1)}$. Then with high probability, for all $0 \leqslant i < t$,*

1. *After the while loop inside the i^{th} iteration terminates, for all f,*

$$\frac{1}{1+\varepsilon/10000}(w_{J_i^{(k)}})_f \leqslant (w_Z)_f \leqslant (1+\varepsilon/10000)(w_{J_i^{(k)}})_f.$$

2. *For all $f \in F_{i+1}^{(k)} \setminus F_i^{(k)}$,*

$$\frac{1}{1+\varepsilon/(10^6 t)}(w_{J_{i+1}^{(k)}})_f \leqslant (w_{H_{i+1}^{(k)}})_f \leqslant (1+\varepsilon/(10^6 t))(w_{J_{i+1}^{(k)}})_f.$$

3. *All edges in $F_{i+1}^{(k)}$ have leverage scores in $J_i^{(k)}$ at least $4\eta_1$.*
4. *$J_{i+1}^{(k)}$ is a $(1+\varepsilon/(10^4 t))$-spectral sparsifier of $J_i^{(k)}$.*

Proof. We prove all statements of this lemma by induction on i. For $i = 0$, since $\delta_f = 0$ for all f, the while loop will not execute. Thus throughout this iteration we have $Z = J_0^{(k)} = G^{(k)}$. This immediately gives 1. By Lemma 4.4.7, the F^* we recover in this iteration contains all edges whose leverage score in $G^{(k)}$ is at least η_1, and all edges in F^* have weight estimates satisfying 2. Since

$H^{(k-1)}$ is a 1.001-spectral sparsifier of $G^{(k-1)}$, and $G^{(k-1)}$ is in turn a 1.001-spectral sparsifier of of $G^{(k)}$, we have that $H^{(k-1)}$ is a 1.003-spectral sparsifier of $G^{(k)}$. As a result, we know that, at the last step of the for loop iteration, all edges with leverage score at least $\geqslant 10\eta_1$ in $G^{(k)}$ will be added to $H^{(k)}$, and all edges added to $H^{(k)}$ have leverage score at least $\geqslant 4\eta_1$ in $G^{(k)}$, so we have 3. This means that $J_1^{(k)}$ is obtained by sampling a set of edges in $J_0^{(k)}$ whose leverage scores in $J_0^{(k)}$ are at most $10\eta_1$ with probability $(1+1/1000)^{-1}$, and multiply their weights by $(1+1/1000)$ if sampled. Using Theorem 2.4.6, we have 4.

We now do an inductive step. Suppose all four statements hold for iterations $0, 1, \ldots, i-1$ where $1 < i < t$. We show that they also hold for iteration i. We first need to analyze the while loop inside iteration i. Let us number a while loop iteration by the value of j at the *end* of the iteration.

Claim 4.4.15. *At the end of while loop iteration j where $j \leqslant t$, we have for all $f \in E_i^{(k)} \cap F_i^{(k)}$*

$$\frac{1}{(1+2\varepsilon_1)^j} \cdot (1+1/1000)^{\delta_f}(w_{J_i^{(k)}})_f \leqslant (w_Z)_f \leqslant (1+2\varepsilon_1)^j(1+1/1000)^{\delta_f}(w_{J_i^{(k)}})_f.$$

Proof. We prove this claim by an induction on j. First we show that the statement is true for $j = 0$ at the beginning of while loop iteration 1. Here all $f \in E_i^{(k)} \cap F_i^{(k)}$ satisfy that $(w_Z)_f = (1+1/1000)^i(w_{G^{(k)}})_f$. Since we set $\delta_f \leftarrow i - c_f$ before the while loop, and by (4.9) $(w_{J_i^{(k)}})_f = (1+1/1000)^{c_f}(w_{G^{(k)}})_f$, we have $(w_Z)_f = (1+1/1000)^{\delta_f}(w_{J_i^{(k)}})_f$, as desired.

Now suppose the statement is true at the end of iteration $j-1$ where $1 < j \leqslant t$. We then show that the statement is also true at the end of iteration j. Let Z_0 be the Z before our updates to Z in iteration j and let Z_1 be the Z after our updates. By Lemma 4.4.7, all edges recovered $f \in F$ have their estimated edge weights $\tilde{w}_f \in [\frac{1}{1+\varepsilon_1}(w_{Z_0})_f, (1+\varepsilon_1)(w_{Z_0})_f]$. Therefore after our updates, we have for any $f \in F$ such that $\delta_f > 0$ that $(w_{Z_1})_f \in [\frac{1}{1+2\varepsilon_1}(1+1/1000)^{-1}(w_{Z_0})_f, (1+2\varepsilon_1)(1+1/1000)^{-1}(w_{Z_0})_f]$, and $(w_{Z_1})_f = (w_{Z_0})_f$ for other edges f. Since we let $\delta_f \leftarrow \delta_f - 1$ for such edges, and do not change the δ_f's of other edges, we have our desired statement for j. □

Claim 4.4.16. *The while loop terminates after at most t iterations.*

Proof. It suffices to show that $\max_f \delta_f$ decreases by 1 in each while loop iteration. Since $\delta_f \leqslant t$ for any f, this will imply that there can be at most t iterations. Then it boils down to showing that for all f^* with $\delta_{f^*} = \max_f \delta_f$, f^* belongs to the recovered edge set F. Since $f^* \in F_i^{(k)}$, by 3 of our induction hypothesis, the leverage score of f^* in $J_{i-1}^{(k)}$ is at least $4\eta_3$. Notice that by 4 of our induction hypothesis, $J_{i-1}^{(k)}$ is a $(1 + 1/1000)$-spectral sparsifier of $G^{(k)}$. Then using the fact that $H^{(k-1)}$ is a 1.003-spectral sparsifier of $G^{(k)}$ (which we proved at the beginning of the proof of this lemma), we have that $H^{(k-1)}$ is a 1.005-spectral sparsifier of $J_{i-1}^{(k)}$.

By Claim 4.4.15, we have at the beginning of each while loop that, for all f,

$$\begin{aligned}(w_Z)_f \in&[\frac{1}{(1+2\varepsilon_1)^t}(1+1/1000)^{\delta_f}(w_{J_i^{(k)}})_f, (1+2\varepsilon_1)^t(1+1/1000)^{\delta_f}(w_{J_i^{(k)}})_f]\\ \subseteq&[\frac{1}{1.01}(1+1/1000)^{\delta_f}(w_{J_i^{(k)}})_f, 1.01(1+1/1000)^{\delta_f}(w_{J_i^{(k)}})_f]. \end{aligned} \tag{4.10}$$

Since $\delta_{f^*} \geqslant \delta_f$ for all f, the above implies

$$L_Z \preceq 1.03(1+1/1000)^{\delta_{f^*}} L_{J_{i-1}^{(k)}}. \tag{4.11}$$

By inverting both sides, we then get

$$L_Z^\dagger \succeq 1.03^{-1}(1+1/1000)^{-\delta_{f^*}} L_{J_{i-1}^{(k)}}^\dagger.$$

Combining this with (4.10), the leverage score of f^* in Z satisfies

$$(w_Z)_{f^*} b_{f^*} L_Z^\dagger b_{f^*} \geqslant \frac{1}{1.01 \cdot 1.03}(w_{J_i^{(k)}})_{f^*} b_{f^*}^T L_{J_{i-1}^{(k)}}^\dagger b_{f^*} \geqslant \frac{4\eta_3}{1.01 \cdot 1.03} \geqslant \eta_3,$$

as desired. □

By Claim 4.4.15, after the while loop terminates, we have that for all f,

$$\begin{aligned}(w_Z)_f \in&[\frac{1}{(1+2\varepsilon_1)^t}(w_{J_i^{(k)}})_f, (1+2\varepsilon_1)^t(w_{J_i^{(k)}})_f]\\ \subseteq&[\frac{1}{(1+\varepsilon/10000)}(w_{J_i^{(k)}})_f, (1+\varepsilon/10000)(w_{J_i^{(k)}})_f],\end{aligned}$$

and thus we have 1. This also implies that Z is a $(1+\varepsilon/10000)$-spectral sparsifier of $J_i^{(k)}$, and as result, for each edge f, its leverage scores in Z and $J_i^{(k)}$ are within a $(1+\varepsilon/10000)^2 < 1.01$ factor of each other.

For all edges in $E_i^{(k)} \setminus F_i^{(k)}$, their weights in Z equal exactly their weights in $J_i^{(k)}$, therefore by Lemma 4.4.7, all edges recovered in F^* not in $F_i^{(k)}$ have weight estimates satisfying 2.

Notice that by 4 of our induction hypothesis, $J_i^{(k)}$ is a $(1+1/1000)$-spectral sparsifier of $G^{(k)}$. Then using the fact that $H^{(k-1)}$ is a 1.003-spectral sparsifier of $G^{(k)}$ (which we proved at the beginning of this proof), we have that $H^{(k-1)}$ is a 1.01-spectral sparsifier of Z. Thus, at the last step of the for iteration, all edges added to $H^{(k)}$ have leverage score at least $\geqslant 5\eta_1$ in Z, and all edges with leverage score $\geqslant 9\eta_1$ in Z will be added to $H^{(k)}$. Thus we also know that all edges added to $H^{(k)}$ have leverage score $\geqslant 4\eta_1$ in $J_i^{(k)}$ (which gives 3), and all edges with leverage score $\geqslant 10\eta_1$ in $J_i^{(k)}$ will be added to $H^{(k)}$.

The above reasoning also implies that $J_{i+1}^{(k)}$ is obtained by sampling a set of edges in $J_i^{(k)}$ whose leverage score is at most $10\eta_1$ with probability $(1+1/1000)^{-1}$, and multiply their weights by $(1+1/1000)$ if sampled. Using Theorem 2.4.6, we have 4. □

Proof of Theorem 4.4.9. **Number of linear measurements.** Note that $(S^{\text{hv}})_{i,j}^{(k)} w_{G^{(k)}[E_i^{(k)}]} \in \mathbb{R}^{n^{6/5}\eta_1^{-1}\varepsilon_1^{-2}\operatorname{polylog}(n,\frac{w_{\max}}{w_{\min}})}$, so the total number of linear measurements is bounded by

$$t^3 n^{6/5}\eta_1^{-1}\varepsilon_1^{-2}\operatorname{polylog}(n,\frac{w_{\max}}{w_{\min}}) \leqslant n^{6/5}\varepsilon^{-4}\operatorname{polylog}(n,\frac{w_{\max}}{w_{\min}},\varepsilon^{-1}).$$

Spectral sparsifier guarantee. By Lemma 4.4.11, $H^{(0)}$ is a 1.001-spectral sparsifier of $G^{(0)}$. We then show that whenever $H^{(k-1)}$ is a 1.001-spectral sparsifier of $G^{(k-1)}$, $H^{(k)}$ is a $(1+\varepsilon/1000)$-spectral sparsifier of $G^{(k)}$ with high probability. Notice that inside each iteration of the outermost

for loop of k, for $i = t$, we have that with high probability $E_t^{(k)} = \emptyset$. This means that $J_t^{(k)}$ consists of solely edges in $F_t^{(k)}$. Thus by Lemma 4.4.14, $H_t^{(k)}$ is a $(1 + \varepsilon/(10^6 t))$-spectral sparsifier of $J_t^{(k)}$. Also by Lemma 4.4.14, $J_t^{(k)}$ is a $(1 + \varepsilon/(10^4 t))^t$-spectral sparsifier of $G^{(k)}$. These combined imply that $H^{(k)}$ is a $(1 + \varepsilon/1000)$-spectral sparsifier of $G^{(k)}$. Now applying an induction on k, we have that $H^{(t)}$ is a $(1 + \varepsilon/1000)$-spectral sparsifier of $G^{(t)}$. Since $G^{(t)}$ is a $(1 + \varepsilon/2)$-spectral sparsifier of G, $H^{(t)}$ is a $(1 + \varepsilon)$-spectral sparsifier of G, as desired. □

4.4.4. Sparsification of G^{sq}

4.4.4.1 Sparsification of G^{sq} by heavy edge recovery

We first give in Figure 4.6 the linear sketch for sparsifying G^{sq} using the recovery of heavy edges in Lemma 4.4.6. We will then prove Lemma 4.4.6 later in Section 4.4.4.2. The ideas for the former linear sketch are the same as the ones we used in Section 4.4.3, since both are about how to sparsify a graph by repeatedly recovering heavy edges.

The performance of the linear sketch is characterized by Lemma 4.4.5.

Lemma 4.4.17. *For any parameter $\varepsilon_2 > 0$ and any integer n, there exists a linear sketch with sketching matrix $S^{\text{sq}} \in \mathbb{R}^{n\varepsilon_2^{-4}\,\text{polylog}(n,\varepsilon_2^{-1},\frac{w_{\max}}{w_{\min}})\times\binom{n}{2}}$ and recovery algorithm* `square-recovery` *such that, given an input graph G of n vertices with weight vector w_G,* `square-recovery`*$(S^{\text{sq}} w_G)$ returns a $(1 + \varepsilon_2)$-spectral sparsifier of G^{sq} with high probability.*

The proof of Lemma 4.4.5 will also be largely similar to that of Theorem 4.4.9 in Section 4.4.3, However, we still include the proof here for completeness.

As in Section 4.4.3, we first prove some useful intermediate lemmas.

Lemma 4.4.18. $(10^6 w_{\max} n^5 K_n)^{\text{sq}}$ *is a* 1.001*-spectral sparsifier of* $(G^{(0)})^{\text{sq}}$.

Proof. Let $\Pi \in \mathbb{R}^{n\times n}$ be the projection matrix on the $n-1$-dimensional subspace orthogonal to the all-one vector. Then Π has $n - 1$ eigenvalues of 1 and one eigenvalue of 0 (with eigenvector being the all-one vector). It is known that $L_{K_n} = n\Pi$, so $L_{(10^6 w_{\max} n^5 K_n)^{\text{sq}}} = 10^{12} w_{\max}^2 n^{11} \Pi$, and has all

`square-sketch`(G, ε_2)

1. Let $t = \left\lceil 10000 \log(10^6 \varepsilon_2^{-1} \frac{w_{\max}}{w_{\min}} \cdot n^{10}) \right\rceil$, and then for each $k = 0, 1, 2, \ldots, t$:
   (a) Let $E_0^{(k)} \supseteq E_1^{(k)} \supseteq \ldots \supseteq E_t^{(k)}$ be subsets of edge slots where $E_0^{(k)} = \binom{V}{2}$ and $E_i^{(k)}$ is obtained by sub-sampling each edge slot in $E_{i-1}^{(k)}$ with probability $(1 + 1/1000)^{-2}$.
   (b) For each pair $0 \leqslant i, j \leqslant t$, let $(S^{\text{sqh}})_{i,j}^{(k)} \in \mathbb{R}^{n\eta_3^{-1}\varepsilon_3^{-2}\,\text{polylog}(n) \times \binom{n}{2}}$ be a sketching matrix in Lemma 4.4.6 with $\eta_3 = \varepsilon_2^2/(10^{12} t^2 \log n)$, $\varepsilon_3 = \varepsilon_2/(10^6 t)$.
2. For each $k, i, j \in [0, t]$, let $G^{(k)} \leftarrow G + (1 + 10^{-4})^{-k} 10^6 w_{\max} n^5 K_n$ and compute the sketch $(S^{\text{sqh}})_{i,j}^{(k)} w_{G^{(k)}[E_i^{(k)}]}$ where $w_{G^{(k)}[E_i^{(k)}]} \in \mathbb{R}^{\binom{n}{2}}$ is weight vector of the subgraph $G^{(k)}[E_i^{(k)}]$.
3. Concatenate these sketches as $S^{\text{sq}} w_G$.

`square-recovery`$(S^{\text{sq}} w_G)$

1. Initially, let $H^{(0)} \leftarrow 10^6 w_{\max} n^5 K_n$ (where $(H^{(0)})^{\text{sq}}$ is a 1.001-spectral sparsifier of $(G^{(0)})^{\text{sq}}$).
2. For $k = 1, 2, \ldots, t$:
   (a) Let $H^{(k)} \leftarrow \emptyset$ (where $(H^{(k)})^{\text{sq}}$ will be a $(1 + \varepsilon_2/1000)$-spectral sparsifier of $(G^{(k)})^{\text{sq}}$).
   (b) Set $c_e \leftarrow 0$ for all $e \in \binom{V}{2}$ (where c_e will be such that e is added to $H^{(k)}$ when $i = c_e$).
   (c) For $i = 0, 1, \ldots, t$:
      i. Let $Z \leftarrow (1 + 1/1000)^i G^{(k)}[E_i^{(k)}] + H^{(k)}[\binom{V}{2} \setminus E_i^{(k)}]$ (Z is to record what graph our linear sketches are taken over).
      ii. Compute sketches $s_j := (S^{\text{sqh}})_{i,j}^{(k)} w_Z, j \in [0, t]$, where w_Z is the weight vector of Z.
      iii. For each $f \in H^{(k)} \cap E_i^{(k)}$, let $\delta_f \leftarrow i - c_f$, and let $\delta_f \leftarrow 0$ for all other edges.
      iv. Let $j \leftarrow 0$. Then while $\exists f : \delta_f > 0$, do the following:
         A. Use Lemma 4.4.6 to recover from s_j and $(H^{(k-1)})^{\text{sq}}$ an edge set F, and $j \leftarrow j + 1$.
         B. For each $f \in F$ such that $\delta_f > 0$, let $\tilde{w}_f$ be the estimate of its weight:
            I. $Z \leftarrow Z - (1 - (1 + 1/1000)^{-1}) \tilde{w}_f f$.
            II. $s_{j'} \leftarrow s_{j'} - (1 - (1 + 1/1000)^{-1})(S^{\text{sqh}})_{i,j'}^{(k)} (\tilde{w}_f \chi_f)$ for all $j \leqslant j' \leqslant t$
            III. $\delta_f \leftarrow \delta_f - 1$.
      v. Use Lemma 4.4.6 to recover from s_t and $(H^{(k-1)})^{\text{sq}}$ a set F^* of edges.
      vi. For each edge $f \in F^*$ with estimated weight $\tilde{w}_f$ such that $\tilde{w}_f^2 b_f^T L^{\dagger}_{(H^{(k-1)})^{\text{sq}}} b_f \geqslant 8\eta_3$ and f is not already in $H^{(k)}$, add f to $H^{(k)}$ with weight $\tilde{w}_f$, and let $c_f \leftarrow i$.
3. Return $(H^{(t)})^{\text{sq}}$.

Figure 4.6: Linear sketch for sparsifying G^{sq}.

$n-1$ nonzero eigenvalues equal to $10^{12}w_{\max}^2 n^{11}$. On the other hand, notice that by expanding

$$\begin{aligned} L_{(G^{(0)})^{\mathrm{sq}}} =& L_{(G+10^6 w_{\max} n^5 K_n)^{\mathrm{sq}}} \\ =& L_{G^{\mathrm{sq}}} + 2\cdot 10^6 w_{\max} n^5 L_G + 10^{12} w_{\max}^2 n^{10} L_{K_n}. \end{aligned}$$

By standard bounds on the largest eigenvalue of the Laplacian matrix of a weighted graph, the largest eigenvalue of an n-vertex graph with maximum weight $w_{\max}$ is at most $nw_{\max}$. Therefore, the largest eigenvalue of $L_{(G^{(0)})^{\mathrm{sq}}-(10^6 w_{\max} n^5 K_n)^{\mathrm{sq}}}$ is at most $3\cdot 10^6 w_{\max}^2 n^6$. This combined with $L_{(G^{(0)})^{\mathrm{sq}}} \succcurlyeq L_{(10^6 w_{\max} n^5 K_n)^{\mathrm{sq}}}$ shows that

$$-\frac{1}{10^5 n^5} L_{(G^{(0)})^{\mathrm{sq}}} \preceq L_{(G^{(0)})^{\mathrm{sq}}} - L_{(10^6 w_{\max} n^5 K_n)^{\mathrm{sq}}} \preceq \frac{1}{10^5 n^5} L_{(G^{(0)})^{\mathrm{sq}}}.$$

This implies that

$$\frac{1}{1.001} L_{(G^{(0)})^{\mathrm{sq}}} \preceq L_{(10^6 w_{\max} n^5 K_n)^{\mathrm{sq}}} \preceq 1.001 L_{(G^{(0)})^{\mathrm{sq}}}$$

as desired. □

Lemma 4.4.19. *For all $k \geqslant 1$, $(G^{(k-1)})^{\mathrm{sq}}$ is a 1.001-spectral sparsifier of $(G^{(k)})^{\mathrm{sq}}$.*

Proof. By definition

$$\begin{aligned} L_{(G^{(k-1)})^{\mathrm{sq}}} =& L_{(G+(1+10^{-4})^{-(k-1)} 10^6 w_{\max} n^5 K_n)^{\mathrm{sq}}} \\ =& L_{G^{\mathrm{sq}}} + 2(1+10^{-4})^{-(k-1)} 10^6 w_{\max} n^5 L_G + (1+10^{-4})^{-2(k-1)} 10^{12} w_{\max}^2 n^{10} L_{K_n} \end{aligned}$$

and

$$\begin{aligned} L_{(G^{(k)})^{\mathrm{sq}}} =& L_{(G+(1+10^{-4})^{-k} 10^6 w_{\max} n^5 K_n)^{\mathrm{sq}}} \\ =& L_{G^{\mathrm{sq}}} + 2(1+10^{-4})^{-k} 10^6 w_{\max} n^5 L_G + (1+10^{-4})^{-2k} 10^{12} w_{\max}^2 n^{10} L_{K_n}. \end{aligned}$$

The second terms of the above two expressions are $(1+10^{-4})$-sparsifiers of each other, and the third

terms of the above two expressions are $(1+10^{-4})^2$-sparsifiers of each other. Therefore, $L_{(G^{(k-1)})^{\mathrm{sq}}}$ is a 1.001-spectral sparsifier of $L_{(G^{(k)})^{\mathrm{sq}}}$. □

Lemma 4.4.20. *$(G^{(t)})^{\mathrm{sq}}$ is a $(1+\varepsilon_2/2)$-spectral sparsifier of G^{sq}.*

Proof. By definition

$$\begin{aligned}
L_{(G^{(t)})^{\mathrm{sq}}} =& L_{(G+(1+10^{-4})^{-t}10^6 w_{\max} n^5 K_n)^{\mathrm{sq}}} \\
=& L_{G^{\mathrm{sq}}} + 2(1+10^{-4})^{-t}10^6 w_{\max} n^5 L_G + (1+10^{-4})^{-2t}10^{12} w_{\max}^2 n^{10} L_{K_n} \\
\preceq& L_{G^{\mathrm{sq}}} + .1\varepsilon_2 \frac{w_{\min}^2}{w_{\max}} n^{-5} L_G + .001\varepsilon_2^2 \frac{w_{\min}^4}{w_{\max}^2} n^{-10} L_{K_n},
\end{aligned}$$

where the last line follows from our choice of t. Thus, the largest eigenvalue of the sum of the last two terms is bounded by $.2\varepsilon_2 w_{\min}^2 n^{-4}$. By standard lower bounds on the second smallest eigenvalue, the second smallest eigenvalue of $L_{G^{\mathrm{sq}}}$ is at least $w_{\min}^2/n^2$. Therefore we have

$$-.2\varepsilon_2 L_{G^{\mathrm{sq}}} \preceq L_{(G^{(t)})^{\mathrm{sq}}} - L_{G^{\mathrm{sq}}} \preceq .2\varepsilon_2 L_{G^{\mathrm{sq}}},$$

which implies that $(G^{(t)})^{\mathrm{sq}}$ is a $(1+\varepsilon_2/2)$-spectral sparsifier of G^{sq}. □

Fix an iteration of the outer for loop of k. Then for an iteration of the inner for loop of i, let $H_i^{(k)}$ be the $H^{(k)}$ at the *beginning* of the iteration, and let $F_i^{(k)}$ be the edges in $H_i^{(k)}$. Define graph

$$J_i^{(k)} := (1+1/1000)^i G^{(k)}[E_i^{(k)} \setminus F_i^{(k)}] + \sum_{\ell=0}^{i-1} (1+1/1000)^\ell G^{(k)}[F_{\ell+1}^{(k)} \setminus F_\ell^{(k)}].$$

Notice that $F_0^{(k)} = \emptyset$ and $J_0^{(k)} = G^{(k)}$. Also by the way we are assigning values to c_f in the algorithm, we have, at the beginning of the for loop (of i) iteration, $f \in F_{c_f+1}^{(k)} \setminus F_{c_f}^{(k)}$ for all $f \in F_i^{(k)}$. Thus we also have

$$J_i^{(k)} := (1+1/1000)^i G^{(k)}[E_i^{(k)} \setminus F_i^{(k)}] + \sum_{f \in F_i^{(k)}} (1+1/1000)^{c_f} (w_{G^{(k)}})_f f. \tag{4.12}$$

Lemma 4.4.21. *Suppose $(H^{(k-1)})^{\mathrm{sq}}$ is a 1.001-spectral sparsifier of $(G^{(k-1)})^{\mathrm{sq}}$. Then with high probability, for all $0 \leqslant i < t$,*

1. *After the while loop inside the i^{th} iteration terminates, for all f,*

$$\frac{1}{1+\varepsilon_2/10000}(w_{J_i^{(k)}})_f \leqslant (w_Z)_f \leqslant (1+\varepsilon_2/10000)(w_{J_i^{(k)}})_f.$$

2. *For all $f \in F_{i+1}^{(k)} \setminus F_i^{(k)}$,*

$$\frac{1}{1+\varepsilon_2/(10^6 t)}(w_{J_{i+1}^{(k)}})_f \leqslant (w_{H_{i+1}^{(k)}})_f \leqslant (1+\varepsilon_2/(10^6 t))(w_{J_{i+1}^{(k)}})_f.$$

3. *All edges in $F_{i+1}^{(k)}$ have leverage scores in $(J_i^{(k)})^{\mathrm{sq}}$ at least $4\eta_3$.*
4. *$(J_{i+1}^{(k)})^{\mathrm{sq}}$ is a $(1+\varepsilon_2/(10^4 t))$-spectral sparsifier of $(J_i^{(k)})^{\mathrm{sq}}$.*

Proof. We prove all statements of this lemma by induction on i. For $i = 0$, since $\delta_f = 0$ for all f, the while loop will not execute. Thus throughout this iteration we have $Z = J_0^{(k)} = G^{(k)}$. This immediately gives 1. Since $(H^{(k-1)})^{\mathrm{sq}}$ is a 1.001-spectral sparsifier of $(G^{(k-1)})^{\mathrm{sq}}$, and $(G^{(k-1)})^{\mathrm{sq}}$ is in turn a 1.001-spectral sparsifier of of $(G^{(k)})^{\mathrm{sq}}$, we have that $(H^{(k-1)})^{\mathrm{sq}}$ is a 1.003-spectral sparsifier of $(G^{(k)})^{\mathrm{sq}}$. Therefore by letting $\tilde{G} = (H^{(k-1)})^{\mathrm{sq}}$, we have

$$\frac{((w_{G^{(k)}})_e b_e^T L_{\tilde{G}}^{\dagger} b_e)^2}{b_e^T L_{\tilde{G}}^{\dagger} L_{(G^{(k)})^{\mathrm{sq}}} L_{\tilde{G}}^{\dagger} b_e} \approx_{1.003^4} \frac{((w_{G^{(k)}})_e b_e^T L_{(G^{(k)})^{\mathrm{sq}}}^{\dagger} b_e)^2}{b_e^T L_{(G^{(k)})^{\mathrm{sq}}}^{\dagger} b_e} = (w_{G^{(k)}})_e^2 b_e^T L_{(G^{(k)})^{\mathrm{sq}}}^{\dagger} b_e, \tag{4.13}$$

where in the first step we have used

$$(w_{G^{(k)}})_e b_e^T L_{\tilde{G}}^{\dagger} b_e \approx_{1.003} (w_{G^{(k)}})_e b_e^T L_{(G^{(k)})^{\mathrm{sq}}}^{\dagger} b_e$$

and

$$b_e^T L_{\tilde{G}}^\dagger L_{(G^{(k)})^{\text{sq}}} L_{\tilde{G}}^\dagger b_e \approx_{1.003} b_e^T L_{\tilde{G}}^\dagger L_{\tilde{G}} L_{\tilde{G}}^\dagger b_e = b_e^T L_{\tilde{G}}^\dagger b_e \approx_{1.003} b_e^T L_{(G^{(k)})^{\text{sq}}}^\dagger b_e.$$

Therefore by Lemma 4.4.6, the F^* we recover in this iteration contains all edges whose leverage score in $(G^{(k)})^{\text{sq}}$ is at least $1.1\eta_3$, and all edges in F^* have weight estimates satisfying 2. Also, at the final step of this for loop iteration, since $(H^{(k-1)})^{\text{sq}}$ is a 1.003-spectral sparsifier of $(G^{(k)})^{\text{sq}}$, all edges with leverage score $\geqslant 10\eta_3$ in $(G^{(k)})^{\text{sq}}$ will be added to $H^{(k)}$, and all edges added to $H^{(k)}$ have leverage score at least $\geqslant 4\eta_3$ in $(G^{(k)})^{\text{sq}}$, so we have 3. This means that $(J_1^{(k)})^{\text{sq}}$ is obtained by sampling a set of edges in $(J_0^{(k)})^{\text{sq}}$ whose leverage scores in $(J_0^{(k)})^{\text{sq}}$ are at most $10\eta_3$ with probability $(1+1/1000)^{-2}$, and multiply their weights by $(1+1/1000)^2$ if sampled. Using Theorem 2.4.6, we have 4.

We now do an inductive step. Suppose all four statements hold for iterations $0, 1, \ldots, i-1$ where $1 < i < t$. We show that they also hold for iteration i. We first need to analyze the while loop inside iteration i. Let us number a while loop iteration by the value of j at the *end* of the iteration.

Claim 4.4.22. *At the end of while loop iteration j where $j \leqslant t$, we have for all $f \in E_i^{(k)} \cap F_i^{(k)}$*

$$\frac{1}{(1+2\varepsilon_3)^j} \cdot (1+1/1000)^{\delta_f} (w_{J_i^{(k)}})_f \leqslant (w_Z)_f \leqslant (1+2\varepsilon_3)^j (1+1/1000)^{\delta_f} (w_{J_i^{(k)}})_f.$$

Proof. We prove this claim by an induction on j. First we show that the statement is true for $j = 0$ at the beginning of while loop iteration 1. Here all $f \in E_i^{(k)} \cap F_i^{(k)}$ satisfy that $(w_Z)_f = (1+1/1000)^i (w_{G^{(k)}})_f$. Since we set $\delta_f \leftarrow i - c_f$ before the while loop, and by (4.12) $(w_{J_i^{(k)}})_f = (1+1/1000)^{c_f} (w_{G^{(k)}})_f$, we have $(w_Z)_f = (1+1/1000)^{\delta_f} (w_{J_i^{(k)}})_f$, as desired.

Now suppose the statement is true at the end of iteration $j-1$ where $1 < j \leqslant t$. We then show that the statement is also true at the end of iteration j. Let Z_0 be the Z before our updates to Z in iteration j and let Z_1 be the Z after our updates. By Lemma 4.4.6, all edges recovered $f \in F$ have their estimated edge weights $\tilde{w}_f \in [\frac{1}{1+\varepsilon_3}(w_{Z_0})_f, (1+\varepsilon_3)(w_{Z_0})_f]$. Therefore after our updates,

we have for any $f \in F$ such that $\delta_f > 0$ that $(w_{Z_1})_f \in [\frac{1}{1+2\varepsilon_3}(1+1/1000)^{-1}(w_{Z_0})_f, (1+2\varepsilon_3)(1+1/1000)^{-1}(w_{Z_0})_f]$, and $(w_{Z_1})_f = (w_{Z_0})_f$ for other edges f. Since we let $\delta_f \leftarrow \delta_f - 1$ for such edges (those with $\delta_f > 0$), and do not change the δ_f's of other edges, we have our desired statement for j. □

Claim 4.4.23. *The while loop terminates after at most t iterations.*

Proof. It suffices to show that $\max_f \delta_f$ decreases by 1 in each while loop iteration. Since $\delta_f \leqslant t$ for any f, this will imply that there can be at most t iterations. Then it boils down to showing that for all f^* with $\delta_{f^*} = \max_f \delta_f$, f^* belongs to the recovered edge set F. Since $f^* \in F_i^{(k)}$, by 3 of our induction hypothesis, the leverage score of f^* in $(J_{i-1}^{(k)})^{\mathrm{sq}}$ is at least $4\eta_3$. Notice that by 4 of our induction hypothesis, $(J_{i-1}^{(k)})^{\mathrm{sq}}$ is a $(1+1/1000)$-spectral sparsifier of $(G^{(k)})^{\mathrm{sq}}$. Then using the fact that $(H^{(k-1)})^{\mathrm{sq}}$ is a 1.003-spectral sparsifier of $(G^{(k)})^{\mathrm{sq}}$ (which we proved at the beginning of the proof of this lemma), we have that $(H^{(k-1)})^{\mathrm{sq}}$ is a 1.005-spectral sparsifier of $(J_{i-1}^{(k)})^{\mathrm{sq}}$.

By Claim 4.4.22, we have at the beginning of each while loop that, for all f,

$$\begin{aligned}(w_Z)_f \in &[\frac{1}{(1+2\varepsilon_3)^t}(1+1/1000)^{\delta_f}(w_{J_i^{(k)}})_f, (1+2\varepsilon_3)^t(1+1/1000)^{\delta_f}(w_{J_i^{(k)}})_f] \\ \subseteq &[\frac{1}{1.01}(1+1/1000)^{\delta_f}(w_{J_i^{(k)}})_f, 1.01(1+1/1000)^{\delta_f}(w_{J_i^{(k)}})_f]. \end{aligned} \tag{4.14}$$

Since $\delta_{f^*} \geqslant \delta_f$ for all f, the above implies

$$L_{Z^{\mathrm{sq}}} \preceq 1.03(1+1/1000)^{2\delta_{f^*}} L_{(J_{i-1}^{(k)})^{\mathrm{sq}}} \preceq 1.04(1+1/1000)^{2\delta_{f^*}} L_{(H^{(k-1)})^{\mathrm{sq}}}, \tag{4.15}$$

where the second inequality follows from that $(H^{(k-1)})^{\mathrm{sq}}$ is a 1.005-spectral sparsifier of $(J_{i-1}^{(k)})^{\mathrm{sq}}$. Let $\tilde{G} = (H^{(k-1)})^{\mathrm{sq}}$. Now in order to show that f^* will be recovered as an edge in F^*, by Lemma 4.4.6, it suffices to show

$$\frac{((w_Z)_{f^*} b_{f^*}^T L_{\tilde{G}}^\dagger b_{f^*})^2}{b_{f^*}^T L_{\tilde{G}}^\dagger L_{Z^{\mathrm{sq}}} L_{\tilde{G}}^\dagger b_{f^*}} \geqslant \eta_3. \tag{4.16}$$

By (4.15), the denominator satisfies

$$b_{f^*}^T L_{\tilde{G}}^\dagger L_{Z^{\mathrm{sq}}} L_{\tilde{G}}^\dagger b_{f^*} \leqslant 1.04(1+1/1000)^{2\delta_{f^*}} b_{f^*}^T L_{\tilde{G}}^\dagger b_{f^*}.$$

Therefore, the LHS of (4.16) is at least

$$(w_Z)_{f^*}^2 (1+1/1000)^{-2\delta_{f^*}} b_{f^*}^T L_{\tilde{G}}^\dagger b_{f^*} \geqslant 1.005^{-1} (w_Z)_{f^*}^2 (1+1/1000)^{-2\delta_{f^*}} b_{f^*}^T L_{(J_{i-1}^{(k)})^{\mathrm{sq}}}^\dagger b_{f^*},$$

where the inequality follows from that $(H^{(k-1)})^{\mathrm{sq}}$ is a 1.005-spectral sparsifier of $(J_{i-1}^{(k)})^{\mathrm{sq}}$. Finally, using (4.14) and that the leverage score of f^* is at least $2\eta_3$ in $(J_{i-1}^{(k)})^{\mathrm{sq}}$, we have that the above is at least η_3, proving (4.16). □

By Claim 4.4.22, after the while loop terminates, we have that for all f,

$$\begin{aligned}(w_Z)_f \in &[\frac{1}{(1+2\varepsilon_3)^t}(w_{J_i^{(k)}})_f, (1+2\varepsilon_3)^t (w_{J_i^{(k)}})_f] \\ \subseteq &[\frac{1}{(1+\varepsilon_2/10000)}(w_{J_i^{(k)}})_f, (1+\varepsilon_2/10000)(w_{J_i^{(k)}})_f],\end{aligned}$$

and thus we have 1. This also implies that Z^{sq} is a $(1+\varepsilon_2/10000)^2$-spectral sparsifier of $(J_i^{(k)})^{\mathrm{sq}}$, and as result, for each edge f, its leverage scores in Z^{sq} and $(J_i^{(k)})^{\mathrm{sq}}$ are within a $(1+\varepsilon_2/10000)^3 < 1.01$ factor of each other.

For all edges in $E_i^{(k)} \setminus F_i^{(k)}$, their weights in Z equal exactly their weights in $J_i^{(k)}$, therefore by Lemma 4.4.6, all edges recovered in F^* not in $F_i^{(k)}$ have weight estimates satisfying 2.

Notice that by 4 of our induction hypothesis, $(J_i^{(k)})^{\mathrm{sq}}$ is a $(1+1/1000)$-spectral sparsifier of $(G^{(k)})^{\mathrm{sq}}$. Then using the fact that $(H^{(k-1)})^{\mathrm{sq}}$ is a 1.003-spectral sparsifier of $(G^{(k)})^{\mathrm{sq}}$ (which we proved at the beginning of this proof), we have that $(H^{(k-1)})^{\mathrm{sq}}$ is a 1.01-spectral sparsifier of Z^{sq}. By letting $\tilde{G} = (H^{(k-1)})^{\mathrm{sq}}$ we have, similar to (4.13),

$$\frac{((w_Z)_e b_e^T L_{\tilde{G}}^\dagger b_e)^2}{b_e^T L_{\tilde{G}}^\dagger L_{Z^{\mathrm{sq}}} L_{\tilde{G}}^\dagger b_e} \approx_{1.01^4} \frac{((w_Z)_e b_e^T L_{Z^{\mathrm{sq}}}^\dagger b_e)^2}{b_e^T L_{Z^{\mathrm{sq}}}^\dagger b_e} = (w_Z)_e^2 b_e^T L_{Z^{\mathrm{sq}}}^\dagger b_e. \tag{4.17}$$

Therefore by Lemma 4.4.6, the F^* we recover in this iteration contains all edges whose leverage scores in $(Z)^{\mathrm{sq}}$ are at least $1.5\eta_3$. Also, at the last step of the for iteration, since $(H^{(k-1)})^{\mathrm{sq}}$ is a 1.01-spectral sparsifier of Z^{sq}, all edges added to $H^{(k)}$ have leverage scores at least $\geqslant 5\eta_3$ in $(Z)^{\mathrm{sq}}$, and all edges with leverage scores $\geqslant 9\eta_3$ in $(Z)^{\mathrm{sq}}$ will be added to $H^{(k)}$. Thus we also know that all edges added to $H^{(k)}$ have leverage scores at least $4\eta_3$ in $(J_i^{(k)})^{\mathrm{sq}}$ (which gives 3), and all edges with leverage scores $\geqslant 10\eta_3$ in $(J_i^{(k)})^{\mathrm{sq}}$ will be added to $H^{(k)}$,

The above reasoning also implies that $(J_{i+1}^{(k)})^{\mathrm{sq}}$ is obtained by sampling a set of edges in $(J_i^{(k)})^{\mathrm{sq}}$ whose leverage score is at most $10\eta_3$ with probability $(1+1/1000)^{-2}$, and multiply their weights by $(1+1/1000)^2$ if sampled. Using Theorem 2.4.6, we have 4. □

Proof of Lemma 4.4.5. **Number of linear measurements.** First, note that $(S^{\mathrm{sqh}})_{i,j}^{(k)} w_{G^{(k)}[E_i^{(k)}]} \in \mathbb{R}^{n\eta_3^{-1}\varepsilon_3^{-2}\operatorname{polylog}(n)}$, so the total number of linear measurements is bounded by

$$t^3 n\eta_3^{-1}\varepsilon_3^{-2}\operatorname{polylog}(n) \leqslant n\varepsilon_2^{-4}\operatorname{polylog}(n, \frac{w_{\max}}{w_{\min}}, \varepsilon_2^{-1}).$$

Spectral sparsifier guarantee. By Lemma 4.4.18, $(H^{(0)})^{\mathrm{sq}}$ is a 1.001-spectral sparsifier of $(G^{(0)})^{\mathrm{sq}}$. We then show that whenever $(H^{(k-1)})^{\mathrm{sq}}$ is a 1.001-spectral sparsifier of $(G^{(k-1)})^{\mathrm{sq}}$, $(H^{(k)})^{\mathrm{sq}}$ is a $(1+\varepsilon_2/1000)$-spectral sparsifier of $(G^{(k)})^{\mathrm{sq}}$ with high probability. Notice that inside each iteration of the outermost for loop of k, for $i = t$, we have that with high probability $E_t^{(k)} = \emptyset$. This means that $J_t^{(k)}$ consists of solely edges in $F_t^{(k)}$. Thus by Lemma 4.4.21, $(H_t^{(k)})^{\mathrm{sq}}$ is a $(1+\varepsilon_2/(10^6 t))^2$-spectral sparsifier of $(J_t^{(k)})^{\mathrm{sq}}$. Also by Lemma 4.4.21, $(J_t^{(k)})^{\mathrm{sq}}$ is a $(1+\varepsilon_2/(10^4 t))^t$-spectral sparsifier of $(G^{(k)})^{\mathrm{sq}}$. These combined imply that $(H^{(k)})^{\mathrm{sq}}$ is a $(1+\varepsilon_2/1000)$-spectral sparsifier of $(G^{(k)})^{\mathrm{sq}}$. Now applying an induction on k, we have that $(H^{(t)})^{\mathrm{sq}}$ is a $(1+\varepsilon_2/1000)$-spectral sparsifier of $(G^{(t)})^{\mathrm{sq}}$. Since $(G^{(t)})^{\mathrm{sq}}$ is a $(1+\varepsilon_2/2)$-spectral sparsifier of G^{sq}, $(H^{(t)})^{\mathrm{sq}}$ is a $(1+\varepsilon_2)$-spectral sparsifier of G^{sq}, as desired. □

4.4.4.2 Recovery of heavy edges in G^{sq}

We now give in Figure 4.7 a linear sketch for recovering heavy edges in G^{sq} and its analysis. This linear sketch is essentially a direct application of ℓ_2-heavy hitters.

`square-heavy-sketch`$(G, \eta_3, \varepsilon_3)$

1. Let $A \in \mathbb{R}^{\eta^{-2}\operatorname{polylog}(n)\times\binom{n}{2}}$ be an ℓ_2-heavy hitter sketching matrix with $\eta = \sqrt{\eta_3}\varepsilon_3/10000$ (Proposition 2.3.12).
2. Let $J \in \mathbb{R}^{O(\delta^{-2}\log n)\times\binom{n}{2}}$ be an ℓ_2-estimation sketching matrix with $\delta = .01$ (Proposition 2.3.16)
3. Concatenate the sketches AW_GB_G and JW_GB_G as $S^{\text{sqh}}w_G$.

`square-heavy-recovery`$(S^{\text{sqh}}w_G, \tilde{G})$

1. For each vertex pair $s, t \in V$:
   (a) Let $x = L^{\dagger}_{\tilde{G}}b_{s,t} \in \mathbb{R}^n$ be the set of vertex potentials induced by a unit electrical flow from s to t in $\tilde{G}$.
   (b) Let $z \in \mathbb{R}^{\binom{n}{2}}$ be an approximation to W_GB_Gx that is recovered from AW_GB_Gx using the recovery algorithm in Proposition 2.3.12.
   (c) Let β be an estimate of $\|W_GB_Gx\|_2$ recovered from JW_GB_Gx using Proposition 2.3.16.
   (d) If $z_{(s,t)} \geqslant .9\sqrt{\eta_3}\beta$, mark edge (s,t) as heavy, and estimate its weight as $\tilde{w}_{(s,t)} = \frac{z_{(s,t)}}{x^Tb_{s,t}}$.
2. Return all edges marked heavy along with the estimates of their weights.

Figure 4.7: Linear sketch for recovering heavy edges in G^{sq}.

The performance of the linear sketch is characterized in Lemma 4.4.6:

Lemma 4.4.24. *For any $\eta_3, \varepsilon_3 \in (0,1)$ and any integer n, there exists a linear sketch with sketching matrix $S^{\text{sqh}} \in \mathbb{R}^{n\eta_3^{-1}\varepsilon_3^{-2}\operatorname{polylog}(n)\times\binom{n}{2}}$ and recovery algorithm `square-heavy-recovery` such that, for an input graph G of n vertices with weight vector w_G and another graph $\tilde{G}$, the subroutine `square-heavy-recovery`$(S^{\text{sqh}}w_G, \tilde{G})$ recovers a set of edges F in G along with estimates of their weights $\tilde{w}_f$'s such that with high probability*

1. *F contains all edges e satisfying*

$$\frac{((w_G)_e b_e^T L^{\dagger}_{\tilde{G}}b_e)^2}{b_e^T L^{\dagger}_{\tilde{G}}L_{G^{\text{sq}}}L^{\dagger}_{\tilde{G}}b_e} \geqslant \eta_3.$$

2. *All edges $f \in F$ satisfy $\frac{1}{1+\varepsilon_3}(w_G)_f \leqslant \tilde{w}_f \leqslant (1+\varepsilon_3)(w_G)_f$.*

Proof of Lemma 4.4.6. **Number of linear measurements.** Notice that the sketches we compute satisfy $AW_GB_G \in \mathbb{R}^{\eta_3^{-1}\varepsilon_3^{-2}\operatorname{polylog}(n)\times n}$ and $JW_GB_G \in \mathbb{R}^{O(\log n)\times n}$. Therefore by concatenating them,

the total number of linear measurements is bounded by $n\eta_3^{-1}\varepsilon_3^{-2}\operatorname{polylog}(n)$.

First guarantee. By the guarantee of the ℓ_2-heavy hitter sketch, we have

$$\begin{aligned}
\left|\left(W_G B_G x\right)_{(s,t)} - z_{(s,t)}\right| \leqslant & \eta \left\|W_G B_G x\right\|_2 \\
\leqslant & (\sqrt{\eta_3}\varepsilon_3/10000)\left\|W_G B_G x\right\|_2 \qquad \text{(by the value of } \eta) \\
= & (\sqrt{\eta_3}\varepsilon_3/10000)\sqrt{b_{s,t}^T L_{\tilde{G}}^{\dagger} L_{G^{\mathrm{sq}}} L_{\tilde{G}}^{\dagger} b_{s,t}}
\end{aligned} \tag{4.18}$$

By the guarantee of the ℓ_2-estimation sketch

$$\beta \in \left[\frac{1}{1.01}\sqrt{b_{s,t}^T L_{\tilde{G}}^{\dagger} L_{G^{\mathrm{sq}}} L_{\tilde{G}}^{\dagger} b_{s,t}}, 1.01\sqrt{b_{s,t}^T L_{\tilde{G}}^{\dagger} L_{G^{\mathrm{sq}}} L_{\tilde{G}}^{\dagger} b_{s,t}}\right]. \tag{4.19}$$

Now fix any e with

$$\frac{((w_e)_G b_e^T L_{\tilde{G}}^{\dagger} b_e)^2}{b_e^T L_{\tilde{G}}^{\dagger} L_{G^{\mathrm{sq}}} L_{\tilde{G}}^{\dagger} b_e} \geqslant \eta_3. \tag{4.20}$$

We then write

$$\begin{aligned}
\left|z_e - \left(W_G B_G x\right)_e\right| \leqslant & \frac{\varepsilon_3}{10000}\sqrt{\eta_3 b_e^T L_{\tilde{G}}^{\dagger} L_{G^{\mathrm{sq}}} L_{\tilde{G}}^{\dagger} b_e} \qquad \text{by (4.18)} \\
\leqslant & \frac{\varepsilon_3}{10000}\sqrt{\eta_3}\sqrt{1/\eta_3} w_e b_e^T L_{\tilde{G}}^{\dagger} b_e \qquad \text{(by (4.20))} \\
\leqslant & .001\varepsilon_3 w_e b_e^T L_{\tilde{G}}^{\dagger} b_e \\
= & .001\varepsilon_3 \left(W_G B_G x\right)_e.
\end{aligned} \tag{4.21}$$

This implies that

$$\begin{aligned}
z_e \geqslant & .999 w_e b_e^T L_{\tilde{G}}^{\dagger} b_e \\
\geqslant & .999\sqrt{\eta_3 b_e^T L_{\tilde{G}}^{\dagger} L_{G^{\mathrm{sq}}} L_{\tilde{G}}^{\dagger} b_e} \qquad \text{(by (4.20))} \\
\geqslant & .999\sqrt{\eta_3}\frac{\beta}{1.01} \qquad \text{(by (4.19))} \\
\geqslant & .9\sqrt{\eta_3}\beta.
\end{aligned}$$

Therefore e will be marked as heavy.

Second guarantee. For e such that $z_e \geqslant .9\sqrt{\eta_3}\beta$, we have

$$\begin{aligned}(W_G B_G x)_e \geqslant & z_e - \frac{\varepsilon_3}{10000}\sqrt{\eta_3 b_e^T L_{\tilde{G}}^\dagger L_{G^{\mathrm{sq}}} L_{\tilde{G}}^\dagger b_e} \qquad \text{(by (4.18))} \\ \geqslant & .9\sqrt{\eta_3}\frac{1}{1.01}\sqrt{b_e^T L_{\tilde{G}}^\dagger L_{G^{\mathrm{sq}}} L_{\tilde{G}}^\dagger b_e} - \frac{\varepsilon_3}{10000}\sqrt{b_e^T L_{\tilde{G}}^\dagger L_{G^{\mathrm{sq}}} L_{\tilde{G}}^\dagger b_e} \qquad \text{(by (4.19))} \\ \geqslant & .87\sqrt{\eta_3 b_e^T L_{\tilde{G}}^\dagger L_{G^{\mathrm{sq}}} L_{\tilde{G}}^\dagger b_e} \qquad \text{(by } \varepsilon_3 < 1\text{)}.\end{aligned}$$

Since $(W_G B_G x)_e = w_e b_e^T L_{\tilde{G}}^\dagger b_e$, we have

$$\frac{((w_e)_G b_e^T L_{\tilde{G}}^\dagger b_e)^2}{b_e^T L_{\tilde{G}}^\dagger L_{G^{\mathrm{sq}}} L_{\tilde{G}}^\dagger b_e} \geqslant .7\eta_3. \tag{4.22}$$

We then write

$$\begin{aligned}|z_e - (W_G B_G x)_e| \leqslant & \frac{\varepsilon_3}{10000}\sqrt{\eta_3 b_{s,t}^T L_{\tilde{G}}^\dagger L_{G^{\mathrm{sq}}} L_{\tilde{G}}^\dagger b_{s,t}} \qquad \text{(by (4.18))} \\ \leqslant & \frac{\varepsilon_3}{10000}\sqrt{\eta_3}\sqrt{1/(.7\eta_3)} w_e b_e^T L_{\tilde{G}}^\dagger b_e \qquad \text{(by (4.22))} \\ \leqslant & .01\varepsilon_3 w_e b_e^T L_{\tilde{G}}^\dagger b_e \\ = & .01\varepsilon_3 (W_G B_G x)_e .\end{aligned} \tag{4.23}$$

By dividing both sides by $x^T b_e$, we have

$$\left|\frac{z_e}{x^T b_e} - \frac{w_e b_e^T L_{\tilde{G}}^\dagger b_e}{x^T b_e}\right| \leqslant .01\varepsilon_3 \frac{w_e b_e^T L_{\tilde{G}}^\dagger b_e}{x^T b_e}.$$

Since $x^T b_e = b_e^T L_{\tilde{G}}^\dagger b_e$, the above is equivalent to $\left|\frac{z_e}{x^T b_e} - w_e\right| \leqslant .01\varepsilon_3 w_e$, and thus we conclude $\frac{1}{1+\varepsilon_3} w_e \leqslant \frac{w_e b_e^T L_{\tilde{G}}^\dagger b_e}{x^T b_e} \leqslant (1+\varepsilon_3) w_e$, as desired. □

4.5. Preliminaries on matrix-weighted graphs

We consider an undirected, $k \times k$ matrix-weighted graph $G = (V, E)$ with $|V| = n$ and $|E| = m$, where each edge (u, v)'s weight is given by $\phi_{uv}\phi_{uv}^T$ for some $\phi_{uv} \in \mathbb{R}^k$. We will assume all $\phi_{uv} \neq 0$,

since one could remove all zero weight edges from E.

Remark 4.5.1 (Block matrices and vectors)**.** *Below, all $nk \times nk$ matrices (or nk-dimensional vectors) are written in block form with block size $k \times k$ (or $k \times 1$). All subscripts also refer to row/column block numbers as opposed to row/column numbers. For example, for an $nk \times nk$ matrix M, we write M_{ij} to denote the $k \times k$ submatrix that is at the intersection of the i^{th} row block and the j^{th} column block.*

Definition 4.5.2 (Degree matrices)**.** For a vertex u, its degree is given by

$$D_u = \sum_{u \sim v} \phi_{uv} \phi_{uv}^T \in \mathbb{R}^{k \times k}.$$

We then define the $nk \times nk$ degree matrix D as a $k \times k$-block diagonal matrix:

$$D = \begin{pmatrix} D_1 & & & \\ & D_2 & & \\ & & \ddots & \\ & & & D_n \end{pmatrix}.$$

Definition 4.5.3. For each edge $(u, v) \in E$, define two nk-dimensional vectors $e_{u \leftarrow v}$ and $e_{v \leftarrow u}$

$$e_{u \leftarrow v} = \begin{pmatrix} 0 \\ \vdots \\ \phi_{uv} \\ 0 \\ \vdots \\ \vdots \end{pmatrix} u^{\text{th}} \text{ block} \qquad \text{and} \qquad e_{v \leftarrow u} = \begin{pmatrix} 0 \\ \vdots \\ \vdots \\ \phi_{uv} \\ 0 \\ \vdots \end{pmatrix} v^{\text{th}} \text{ block}$$

Proposition 4.5.4. $D = \sum_{u \sim v} (e_{u \leftarrow v} e_{u \leftarrow v}^T + e_{v \leftarrow u} e_{v \leftarrow u}^T)$.

When defining almost regular graphs, we will be interested in the quantities $\phi_{uv}^T D_u^\dagger \phi_{uv}$. We will call $\phi_{uv}^T D_u^\dagger \phi_{uv}$ the leverage score of edge (u, v) with respect to vertex u.

Definition 4.5.5 (Leverage score of an edge w.r.t. a vertex)**.** We call $\phi_{uv}^T D_u^\dagger \phi_{uv}$ *the leverage score of edge* (u, v) *w.r.t. vertex* u*.*

Proposition 4.5.6. *For any edge* (u, v)*,* $\phi_{uv}^T D_u^\dagger \phi_{uv} = e_{u \leftarrow v}^T D^\dagger e_{u \leftarrow v}$*.*

Definition 4.5.7 (Adjacency matrices)**.** The $nk \times nk$ adjacency matrix A is given by $A_{uv} = \phi_{uv}\phi_{uv}^T$ for $u \sim v$ and $A_{uv} = 0$ otherwise.

In the following definitions, we assume there is a (arbitrarily) fixed orientation of all edges.

Definition 4.5.8 (Incidence vectors)**.** For an edge (u, v) oriented as $u \to v$, its nk-dimensional incidence vector b_{uv} is given by

$$b_{uv} = \begin{pmatrix} 0 \\ \vdots \\ \phi_{uv} \\ 0 \\ \vdots \\ -\phi_{uv} \\ 0 \\ \vdots \end{pmatrix} \begin{matrix} \\ \\ u^{\text{th}} \text{ block} \\ \\ \\ v^{\text{th}} \text{ block} \\ \\ \end{matrix} .$$

Definition 4.5.9 (Laplacian matrices)**.** The Laplacian matrix is given by $L = D - A$.

Proposition 4.5.10. *We have* $L = \sum_{u \sim v} b_{uv} b_{uv}^T$*.*

Proposition 4.5.11. *For any vector* $x \in \mathbb{R}^{nk}$*, we have*

$$x^T L x = \sum_{u \sim v} \langle x_u - x_v, \phi_{uv} \rangle^2 .$$

Definition 4.5.12 (Projection matrices)**.** For an s-dimensional subspace $\mathcal{S} \subseteq \mathbb{R}^d$ with orthonormal

basis $b_1, \ldots, b_s \in \mathbb{R}^d$, we write

$$\Pi_{\mathcal{S}} = \sum_{i=1}^{s} b_i b_i^T \in \mathbb{R}^{d \times d}$$

to denote the projection matrix onto $\mathcal{S}$. We will also write, for a symmetric matrix $M \in \mathbb{R}^{d \times d}$, $\Pi_M \in \mathbb{R}^{d \times d}$ to denote the projection matrix onto the range of M.

Definition 4.5.13 (Normalized Laplacians)**.** The normalized Laplacian matrix is given by

$$N \stackrel{\text{def}}{=} D^{\dagger/2} L D^{\dagger/2} = \Pi_D - D^{\dagger/2} A D^{\dagger/2}.$$

Proposition 4.5.14. $N = \sum_{u \sim v} (D^{\dagger/2} b_{uv})(D^{\dagger/2} b_{uv})^T$.

Proposition 4.5.15. *For any vector $x \in \mathbb{R}^{nk}$, we have*

$$x^T N x = \sum_{u \sim v} \left(\left\langle x_u, D_u^{\dagger/2} \phi_{uv} \right\rangle - \left\langle x_v, D_v^{\dagger/2} \phi_{uv} \right\rangle \right)^2.$$

Proposition 4.5.16. *The eigenvalues of N is between $[0, 2]$.*

Sometimes, instead of using exact degrees, we will normalize the Laplacian matrix by approximate degrees. Thus we also define $\tilde{D}$-normalized Laplacian matrix where $\tilde{D} \in \mathbb{R}^{nk \times nk}$ is a semidefinite, $k \times k$-block diagonal matrix that has the same range as D.

Definition 4.5.17 ($\tilde{D}$-normalized Laplacians)**.** For a semidefinite, $k \times k$-block diagonal matrix $\tilde{D} \in \mathbb{R}^{nk \times nk}$ whose range is the same as D, the $\tilde{D}$-normalized Laplacian matrix is given by

$$\tilde{N} \stackrel{\text{def}}{=} \tilde{D}^{\dagger/2} L \tilde{D}^{\dagger/2}.$$

Proposition 4.5.18. $\tilde{N} = \sum_{u \sim v} (\tilde{D}^{\dagger/2} b_{uv})(\tilde{D}^{\dagger/2} b_{uv})^T$.

Proposition 4.5.19. *For $\tilde{D}$ satisfying*

$$\frac{1}{\kappa} D \preceq \tilde{D} \preceq \kappa D$$

for some $\kappa > 1$, the eigenvalues of $\tilde{N} = \tilde{D}^{\dagger/2} L \tilde{D}^{\dagger/2}$ is between $[0, 2\kappa]$.

Note that, since the degree matrix D is not necessarily full-rank, for either L, N, or $\tilde{N}$, there will always be some "trivial" zero eigenvalues that correspond to the kernel of D. Therefore we shall focus only on the remaining "nontrivial" eigenvalues and their eigenvectors.

Definition 4.5.20 (Nontrivial eigenvalues and eigenvectors)**.** An eigenvalue λ and its associated eigenvector $f \in \mathbb{R}^{nk}$ of either L, N, or $\tilde{N}$ are said to be *nontrivial* if we have

$$f_u \perp \ker(D_u), \ \forall u \in V.$$

Proposition 4.5.21. *The number of nontrivial eigenvalues (counted with multiplicity) of either L, N, or $\tilde{N}$ is exactly* $\operatorname{rank}(D)$.

4.6. Almost regular graphs have only few small eigenvalues

We first give the definition of almost regularity in matrix-weighted graphs.

Definition 4.6.1 (Almost regular graphs)**.** Let $G = (V, E)$ be a $k \times k$ matrix-weighted graph with edge weights $\phi_{uv}\phi_{uv}^T$. For a $\gamma \geqslant 1$, we say G is γ-almost regular if for all $u \in V$ and $(u, v) \in E$, we have

$$\phi_{uv}^T D_u^\dagger \phi_{uv} \leqslant \frac{\gamma \cdot k}{n}. \tag{4.24}$$

We then consider the spectrum of the $\tilde{D}$-normalized Laplacian (Definition 4.5.17) of a graph G for some semidefinite, $k \times k$-block diagonal matrix $\tilde{D}$ with the same range as D. Specifically, we show that when G is almost regular and $\tilde{D}$ is close to D, the number of small nontrivial eigenvalues (Definition 4.5.20) is small.

Theorem 4.6.2. *Let $G = (V, E)$ be a $k \times k$ matrix-weighted, γ-almost regular graph. Suppose $\tilde{D}$ is a semidefinite, $k \times k$-block diagonal matrix satisfying that*

$$\frac{1}{\kappa} D \preceq \tilde{D} \preceq \kappa D$$

for some $\kappa \geqslant 1$. Then for any $\delta \in (0,1)$, the number of nontrivial eigenvalues of $\tilde{N} = \tilde{D}^{\dagger/2} L \tilde{D}^{\dagger/2}$ that are at most $\frac{1}{\kappa} - \delta$ is at most $\frac{\gamma \kappa^2 k^2}{\delta^2}$.

This theorem will be a consequence a lemma that characterizes certain properties of the "spectral embedding" induced by the bottom eigenvectors of $\tilde{N}$. We note that similar spectral embeddings were previously used to prove higher-order Cheeger inequalities for scalar-weighted graphs [LRTV12, LGT14].

Definition 4.6.3 (Spectral embeddings)**.** Given orthonormal vectors $f_1, \ldots, f_\ell \in \mathbb{R}^{nk}$, define an $\ell \times nk$ matrix $\mathcal{F}$ whose rows are transposes of $f_1, \ldots, f_\ell$:

$$\mathcal{F} = \begin{array}{|c|c|c|c|c|} \hline (f_1)_1^T & \cdots & (f_1)_u^T & \cdots & (f_1)_n^T \\ \hline (f_2)_1^T & \cdots & (f_2)_u^T & \cdots & (f_2)_n^T \\ \hline \multicolumn{5}{|c|}{\vdots} \\ \hline (f_\ell)_1^T & \cdots & (f_\ell)_u^T & \cdots & (f_\ell)_n^T \\ \hline \end{array} \in \mathbb{R}^{\ell \times nk}.$$

Then define an embedding $F : V \to \mathbb{R}^{\ell \times k}$ by letting $F(u)$ equal the u^{th} column block of $\mathcal{F}$:

$$F(u) = \begin{array}{|c|} \hline (f_1)_u^T \\ \hline (f_2)_u^T \\ \hline \vdots \\ \hline (f_\ell)_u^T \\ \hline \end{array} \in \mathbb{R}^{\ell \times k}.$$

We call F the spectral embedding induced by $f_1, \ldots, f_\ell$, and $\mathcal{F}$ the embedding matrix induced by $f_1, \ldots, f_\ell$.

Essentially, the following lemma says that the spectral embedding induced by the (nontrivial) bottom eigenvectors has norm spread out across a large number of vertices.

Lemma 4.6.4. *Let $G = (V, E)$ be a $k \times k$ matrix-weighted, γ-almost regular graph. Suppose $\tilde{D}$ is*

a semidefinite, $k \times k$-block diagonal matrix satisfying that

$$\frac{1}{\kappa} D \preceq \tilde{D} \preceq \kappa D$$

for some $\kappa \geqslant 1$. Fix a $\delta \in (0,1)$ and let $0 \leqslant \tilde{\lambda}_1 \leqslant \ldots \leqslant \tilde{\lambda}_\ell \leqslant \frac{1}{\kappa} - \delta$ be all nontrivial eigenvalues of $\tilde{N} = \tilde{D}^{\dagger/2} L \tilde{D}^{\dagger/2}$ that are $\leqslant \frac{1}{\kappa} - \delta$. Let $\tilde{f}_1, \ldots, \tilde{f}_\ell$ be a corresponding set of orthonormal, nontrivial eigenvectors, which by the definition of nontriviality satisfies

$$(\tilde{f}_i)_u \perp \ker(D_u), \ \forall i \in [\ell], u \in V.$$

Let $\tilde{F}$ be the spectral embedding induced by $\tilde{f}_1, \ldots, \tilde{f}_\ell$. Then we have for all $u \in V$

$$\lambda_{\max}\left(\tilde{F}(u)^T \tilde{F}(u)\right) \leqslant \frac{\gamma \kappa^2 k}{\delta^2 n}, \tag{4.25}$$

where $\lambda_{\max}(\cdot)$ denotes taking the largest eigenvalue.

Proof of Theorem 4.6.2 using Lemma 4.6.4. Let ℓ be the number of nontrivial eigenvalues of $\tilde{N}$ that are $\leqslant \frac{1}{\kappa} - \delta$ and let $\tilde{f}_1, \ldots, \tilde{f}_\ell$ be a corresponding set of orthonormal, nontrivial eigenvectors. Let $\tilde{F}$ be the spectral embedding induced by $\tilde{f}_1, \ldots, \tilde{f}_\ell$. Since all $\tilde{f}_i$'s are unit vectors, we have

$$\sum_u \mathrm{Tr}\left(\tilde{F}(u)^T \tilde{F}(u)\right) = \sum_u \left\|\tilde{F}(u)\right\|_F^2 = \sum_u \sum_{i=1}^{\ell} \left\|(\tilde{f}_i)_u\right\|^2 = \ell.$$

Therefore there must exist a vertex u with

$$\mathrm{Tr}\left(\tilde{F}(u)^T \tilde{F}(u)\right) \geqslant \frac{\ell}{n}.$$

As $\tilde{F}(u)^T \tilde{F}(u)$ is a $k \times k$ positive semidefinite matrix, we have

$$\lambda_{\max}\left(\tilde{F}(u)^T \tilde{F}(u)\right) \geqslant \frac{1}{k} \cdot \mathrm{Tr}\left(\tilde{F}(u)^T \tilde{F}(u)\right) \geqslant \frac{\ell}{nk}.$$

On the other hand, by Lemma 4.6.4, $\lambda_{\max}\left(\tilde{F}(u)^T \tilde{F}(u)\right) \leqslant \frac{\gamma \kappa^2 k}{\delta^2 n}$. Thus we must have $\ell \leqslant \frac{\gamma \kappa^2 k^2}{\delta^2}$. $\square$

Proof of Lemma 4.6.4. Let $\tilde{\mathcal{F}}$ be the embedding matrix induced by $\tilde{f}_1, \ldots, \tilde{f}_\ell$. Suppose for the sake of contradiction (4.25) is violated by some $u \in V$. That is, we have

$$\lambda_{\max}\left(\tilde{F}(u)^T \tilde{F}(u)\right) > \frac{\gamma \kappa^2 k}{\delta^2 n}. \tag{4.26}$$

Define a diagonal matrix $\tilde{\Lambda} \in \mathbb{R}^{\ell \times \ell}$ by

$$\tilde{\Lambda} = \begin{pmatrix} \tilde{\lambda}_1 & & \\ & \ddots & \\ & & \tilde{\lambda}_\ell \end{pmatrix} \in \mathbb{R}^{\ell \times \ell}.$$

Then, since $\tilde{\lambda}_1, \ldots, \tilde{\lambda}_\ell$ and $\tilde{f}_1, \ldots, \tilde{f}_\ell$ are corresponding eigenvalues and eigenvectors of $\tilde{N}$, we immediately have (recall that $\tilde{\mathcal{F}}$'s rows are $\tilde{f}_1^T, \ldots, \tilde{f}_\ell^T$)

$$\tilde{\Lambda}\tilde{\mathcal{F}} = \tilde{\mathcal{F}}\tilde{N},$$

and in particular, by restricting to the u^{th} column block on both sides,

$$\tilde{\Lambda}\tilde{F}(u) = \tilde{F}(u)\tilde{D}_{uu}^{\dagger/2} D_u \tilde{D}_{uu}^{\dagger/2} - \sum_{u \sim v} \tilde{F}(v)\tilde{D}_{vv}^{\dagger/2} \phi_{uv} \phi_{uv}^T \tilde{D}_{uu}^{\dagger/2}. \tag{4.27}$$

Now consider the $\ell \times \ell$ matrix $\tilde{F}(u)\tilde{F}(u)^T$, whose largest eigenvalue is equal to $\lambda_{\max}(\tilde{F}(u)^T\tilde{F}(u))$. Let $g \in \mathbb{R}^\ell$ be a unit eigenvector corresponding to $\lambda_{\max}(\tilde{F}(u)\tilde{F}(u)^T)$. Since $\tilde{f}_1, \ldots, \tilde{f}_\ell$ are orthonormal, we have

$$\begin{aligned} \sum_{v \in V} g^T \tilde{F}(v)\tilde{F}(v)^T g =& g^T \left(\sum_{v \in V} \tilde{F}(v)\tilde{F}(v)^T\right) g \\ =& g^T \left(\tilde{\mathcal{F}}\tilde{\mathcal{F}}^T\right) g \\ =& g^T I g \\ =& \|g\|^2 = 1. \end{aligned} \tag{4.28}$$

We will then show a contradiction by arguing that (4.27) cannot be true given (4.26). First by multiplying g^T to the left and $\tilde{F}(u)^T g$ to the right to both sides of (4.27) we have

$$g^T \tilde{\Lambda} \tilde{F}(u) \tilde{F}(u)^T g = g^T \tilde{F}(u) \tilde{D}_{uu}^{\dagger/2} D_u \tilde{D}_{uu}^{\dagger/2} \tilde{F}(u)^T g - \sum_{u \sim v} g^T \tilde{F}(v) \tilde{D}_{vv}^{\dagger/2} \phi_{uv} \phi_{uv}^T \tilde{D}_{uu}^{\dagger/2} \tilde{F}(u)^T g. \tag{4.29}$$

Using the fact that g is a unit eigenvector corresponding to $\lambda_{\max}(\tilde{F}(u)\tilde{F}(u)^T)$, we can upper bound the LHS of (4.29) by

$$\begin{aligned} g^T \tilde{\Lambda} \tilde{F}(u) \tilde{F}(u)^T g =& \lambda_{\max} \left(\tilde{F}(u) \tilde{F}(u)^T \right) \cdot g^T \tilde{\Lambda} g \\ \leqslant& \left(\frac{1}{\kappa} - \delta \right) \cdot \lambda_{\max} \left(\tilde{F}(u) \tilde{F}(u)^T \right), \end{aligned}$$

where the last inequality holds since all eigenvalues $\tilde{\lambda}_1, \ldots, \tilde{\lambda}_\ell \leqslant \frac{1}{\kappa} - \delta$. We can also lower bound the first term on the RHS of (4.29) by

$$g^T \tilde{F}(u) \tilde{D}_{uu}^{\dagger/2} D_u \tilde{D}_{uu}^{\dagger/2} \tilde{F}(u)^T g \geqslant \frac{1}{\kappa} \cdot g^T \tilde{F}(u) \tilde{F}(u)^T g = \frac{1}{\kappa} \cdot \lambda_{\max}(\tilde{F}(u) \tilde{F}(u)^T),$$

where the first inequality follows from $\tilde{D} \preceq \kappa D$. Therefore, in order to contradict (4.29), it suffices to show that the second term on the RHS of (4.29) satisfies

$$\sum_{u \sim v} g^T \tilde{F}(v) \tilde{D}_{vv}^{\dagger/2} \phi_{uv} \phi_{uv}^T \tilde{D}_{uu}^{\dagger/2} \tilde{F}(u)^T g \; < \; \delta \cdot \lambda_{\max} \left(\tilde{F}(u) \tilde{F}(u)^T \right).$$

We then do so by applying the Cauchy-Schwarz inequality:

$$
\begin{aligned}
&\sum_{u\sim v} g^T\tilde{F}(v)\tilde{D}_{vv}^{\dagger/2}\phi_{uv}\phi_{uv}^T\tilde{D}_{uu}^{\dagger/2}\tilde{F}(u)^T g \\
\leqslant&\sqrt{\sum_{u\sim v} g^T\tilde{F}(v)\tilde{F}(v)^T g}\sqrt{\sum_{u\sim v} g^T\tilde{F}(u)\tilde{D}_{uu}^{\dagger/2}\phi_{uv}\phi_{uv}^T\tilde{D}_{vv}^{\dagger}\phi_{uv}\phi_{uv}^T\tilde{D}_{uu}^{\dagger/2}\tilde{F}(u)^T g} \\
\leqslant&1\cdot\sqrt{\sum_{u\sim v} g^T\tilde{F}(u)\tilde{D}_{uu}^{\dagger/2}\phi_{uv}\phi_{uv}^T\tilde{D}_{vv}^{\dagger}\phi_{uv}\phi_{uv}^T\tilde{D}_{uu}^{\dagger/2}\tilde{F}(u)^T g} \qquad \text{(by (4.28))} \\
\leqslant&\sqrt{\frac{\gamma\kappa\cdot k}{n}}\cdot\sqrt{\sum_{u\sim v} g^T\tilde{F}(u)\tilde{D}_{uu}^{\dagger/2}\phi_{uv}\phi_{uv}^T\tilde{D}_{uu}^{\dagger/2}\tilde{F}(u)^T g} \\
&\qquad \text{(since } \phi_{uv}^T\tilde{D}_{vv}^{\dagger}\phi_{uv}\leqslant\frac{\gamma\kappa\cdot k}{n} \text{ by } \gamma\text{-almost regularity and } \tilde{D}\succeq\frac{1}{\kappa}D\text{)} \\
=&\sqrt{\frac{\gamma\kappa\cdot k}{n}}\sqrt{g^T\tilde{F}(u)\tilde{D}_{uu}^{\dagger/2}\left(\sum_{u\sim v}\phi_{uv}\phi_{uv}^T\right)\tilde{D}_{uu}^{\dagger/2}\tilde{F}(u)^T g} \\
=&\sqrt{\frac{\gamma\kappa\cdot k}{n}}\sqrt{g^T\tilde{F}(u)\tilde{D}_{uu}^{\dagger/2}D_u\tilde{D}_{uu}^{\dagger/2}\tilde{F}(u)^T g} \\
\leqslant&\sqrt{\frac{\gamma\kappa\cdot k}{n}}\sqrt{\kappa}\sqrt{g^T\tilde{F}(u)\tilde{F}(u)^T g} \qquad \text{(by } \tilde{D}\succeq\frac{1}{\kappa}D\text{)} \\
=&\sqrt{\frac{\gamma\kappa^2 k}{n}}\sqrt{\lambda_{\max}(\tilde{F}(u)\tilde{F}(u)^T)} \\
<&\delta\cdot\lambda_{\max}(\tilde{F}(u)\tilde{F}(u)^T) \qquad \text{(since by our assumption (4.26) } \lambda_{\max}(\tilde{F}(u)^T\tilde{F}(u))>\frac{\gamma\kappa^2 k}{\delta^2 n}\text{).}
\end{aligned}
$$

Therefore it must be the case that all $\lambda_{\max}(\tilde{F}(u)^T\tilde{F}(u))\leqslant\frac{\gamma\kappa^2 k}{\delta^2 n}$, as desired □

4.7. Almost regular graph decomposition

In this section, we show that every matrix-weighted graph that is sufficiently dense can be made into an almost regular graph (in the sense of Definition 4.6.1) by downscaling a small number of edges.

Let us first introduce some notations for rescaled graphs. For a $k\times k$ matrix-weighted graph $G=(V,E)$ with edge weights $\phi_{uv}\phi_{uv}^T$'s and a scaling $s:E\to[0,1]$, we will write G^s to denote the graph obtained from G by rescaling each edge (u,v)'s weight to $(s_{uv}\phi_{uv})(s_{uv}\phi_{uv})^T$. For simplicity we will use the superscript s when dealing with vectors and matrices associated with G^s. For example, we have $\phi_{uv}^s=s_{uv}\phi_{uv}$, $D_u^s=\sum_{u\sim v}(s_{uv}\phi_{uv})(s_{uv}\phi_{uv})^T$, and $L^s=\sum_{(u,v)\in E}s_{uv}^2 b_{uv}b_{uv}^T$. Similarly, for

a subset of edges $F \subseteq E$, we also use the superscript F to denote matrices associated with the induced subgraph $G[F]$, and thus, for example, $D_u^F = \sum_{(u,v)\in F} \phi_{uv}\phi_{uv}^T$.

Our main result in this section is a deterministic algorithm for finding a large almost regular subgraph within any given graph that is sufficiently dense.

Theorem 4.7.1. *There is a deterministic algorithm* `almost-regular-decomp` *that, given any $k \times k$ matrix-weighted graph $G = (V, E)$ with edge weights $\phi_{uv}\phi_{uv}^T$'s and any $\gamma \geqslant 1$, outputs a scaling $s : E \to [0, 1]$ such that*

1. *The rescaled graph G^s is γ-almost regular:*

$$(\phi_{uv}^s)^T \left(D_u^s\right)^\dagger \phi_{uv}^s \leqslant \frac{\gamma \cdot k}{n},\ \forall (u, v) \in E.$$

2. *The number of edges $(u, v) \in E$ with $s_{uv} \in (0, 1)$ is at most $\frac{8n^2}{\gamma}$.*
3. *The number of edges $(u, v) \in E$ with $s_{uv} = 0$ is at most $\frac{8n}{\gamma \cdot k}(\text{rank}\,(D) - \text{rank}\,(D^s))$.*

The algorithm terminates in finite time.

Note that since $\text{rank}\,(D) \leqslant nk$, the number of edges with $s_{uv} = 0$ is at most $\frac{8n^2}{\gamma}$. Therefore, by setting γ to be $\geqslant \frac{32n^2}{|E|}$, the total fraction of edges with $s_{uv} < 1$ is no more than $1/2$. Also note that our goal is to prove the existence of such a scaling for any given weights $\phi_{uv}\phi_{uv}^T$'s that can potentially have *infinite* precision, so we only focus on designing an algorithm that terminates in finite time, as opposed to giving an explicit bound on its running time.

Our approach for proving Theorem 4.7.1 is motivated by the one in [CLM+15], using which the authors showed that given a set of vectors, one can, by downscaling a small number of them, make every vector have small leverage score compared to the average. The main difference between our result and theirs is that we have an additional bound on the number of completely deleted edges that is proportional to the rank reduction (the third guarantee in Theorem 4.7.1), which will be useful when we incorporate Theorem 4.7.1 into our expander decomposition later in Section 4.8.

We now give the pseudocodes of our algorithms `almost-regular-decomp` and `whack-a-mole-I` below. At a high level, in `almost-regular-decomp`, we first try to eliminate edges with high leverage

scores[9], by iteratively halving the weight of any such edge for some large number of iterations. The latter iterative process is essentially what `whack-a-mole-I` does. Then after `whack-a-mole-I` terminates, either (i) all edges have small leverage scores, in which case we are done, or (ii) the edge weights have become sufficiently divergent so that we can decrease the rank of D by deleting a small number of edges. If it is the second case, we then delete those edges and restart the process from beginning. We will show that the total number of deleted/rescaled edges is small, so that we end up with a large almost regular graph.

Algorithm 4.7.1 `whack-a-mole-I`(G, γ, T, s^0)

Input: G: a $k \times k$ matrix-weighted graph $G = (V, E)$ with edge weights $\phi_{uv}\phi_{uv}^T$.
γ: an almost regularity parameter.
T: an integer denoting the iteration count.
s^0: an initial scaling, mapping from $E \to [0, 1]$.
Output: s: a scaling, mapping from $E \to [0, 1]$.
Procedure:

Let $s \leftarrow s^0$.
for $t \leftarrow 1$ to T **do**
    if $\exists u \in V, (u, v) \in E$ with $(\phi_{uv}^s)^T (D_u^s)^\dagger \phi_{uv}^s > \frac{\gamma \cdot k}{n}$ **then**
        Pick an arbitrary pair of such u and (u, v).
        $s_{uv} \leftarrow s_{uv}/2$.
return s.

Analysis of Algorithm 4.7.1. We first show that after `whack-a-mole-I` terminates, all edges that have been downscaled have large leverage scores w.r.t. at least one of their endpoints.

Lemma 4.7.2. *Suppose the initial scaling s^0 satisfies that $s_{uv}^0 \in \{0, 1\}$ for all $(u, v) \in E$. Then after `whack-a-mole-I` terminates, for any $(u, v) \in E$ such that $s_{uv} \in (0, 1)$, the leverage score of (u, v) w.r.t. at least one of u, v is $\geqslant \frac{\gamma \cdot k}{4n}$. That is, at least one of the following two statements is true:*

1. $(\phi_{uv}^s)^T (D_u^s)^\dagger \phi_{uv}^s \geqslant \frac{\gamma \cdot k}{4n}$.

[9]Recall that in Definition 4.5.5 we call the quantity $(\phi_{uv})^T (D_u)^\dagger \phi_{uv}$ the leverage of edge (u, v) w.r.t. vertex u.

Algorithm 4.7.2 `almost-regular-decomp`(G, γ)

Input: G: a $k \times k$ matrix-weighted graph $G = (V, E)$ with edge weights $\phi_{uv}\phi_{uv}^T$.
γ: an almost regularity parameter.
Output: s: a scaling, mapping from $E \to [0, 1]$.
Procedure:

Initially, let $s_{uv} \leftarrow 1$ for all $(u, v) \in E$.
Let $\ell \leftarrow \min_{(u,v)\in E} \|\phi_{uv}\|^2$ and $r \leftarrow \max_{(u,v)\in E} \|\phi_{uv}\|^2$.
Let $\alpha \leftarrow \max\left\{8n^4k,\ 4nk \cdot r \cdot \max_{F\subseteq E} \mu_{\max}\left(\left(D^F\right)^\dagger\right)\right\}$.
Let $T \leftarrow n^2(4n^2k \log \alpha + \log(r/\ell))$.
while $\text{rank}\,(D^s) > 0$ **do**
    $s \leftarrow$ `whack-a-mole-I`(G, γ, T, s).
    if $\forall u \in V, (u, v) \in E$ we have $(\phi_{uv}^s)^T (D_u^s)^\dagger \phi_{uv}^s \leqslant \frac{\gamma \cdot k}{n}$ **then**
        Return s and halt.
    else
        Compute all eigenvalues of D^s and let $0 < \mu_1 \leqslant \ldots \leqslant \mu_p$ be the positive ones.
        Let $\ell^s \leftarrow \min_{(u,v)\in E, s_{uv}>0} \|\phi_{uv}^s\|^2$
        Find the smallest $\rho \geqslant \ell^s$ such that all $\|\phi_{uv}^s\|^2$ and μ_i fall in $\mathbb{R} \setminus (\rho, \rho \cdot \alpha)$.
        Set $s_{uv} \leftarrow 0$ for all $(u, v) \in E$ with $\|\phi_{uv}^s\|^2 \leqslant \rho$.
        Set $s_{uv} \leftarrow 1$ for all $(u, v) \in E$ with $s_{uv} > 0$.
return s.

2. $(\phi_{uv}^s)^T (D_v^s)^\dagger \phi_{uv}^s \geqslant \frac{\gamma \cdot k}{4n}$.

Proof. Fix an edge $(u, v) \in E$ for which $s_{uv} \in (0, 1)$ after `whack-a-mole-I` terminates. Since initially $s_{uv}^0 \in \{0, 1\}$, it must be the case that $s_{uv}^0 = 1$, and thus s_{uv} must change at least once during the process. Let $t_{uv} \in [1, T]$ be the last for loop iteration in which s_{uv} changes. Then we know that at the beginning of iteration t_{uv}, we have either $(\phi_{uv}^s)^T (D_u^s)^\dagger \phi_{uv}^s > \frac{\gamma \cdot k}{n}$ or $(\phi_{uv}^s)^T (D_v^s)^\dagger \phi_{uv}^s > \frac{\gamma \cdot k}{n}$. Suppose w.l.o.g. it is the first case. We first show that after we set $s_{uv} \leftarrow s_{uv}/2$, we have $(\phi_{uv}^s)^T (D_u^s)^\dagger \phi_{uv}^s \geqslant \frac{\gamma \cdot k}{4n}$. For clarity let us write s and s' to denote the scalings before and after the update in iteration t_{uv} respectively. Then we have

$$\begin{aligned}(\phi_{uv}^{s'})^T (D_u^{s'})^\dagger \phi_{uv}^{s'} \geqslant & (\phi_{uv}^{s'})^T (D_u^s)^\dagger \phi_{uv}^{s'} \qquad (D_u^{s'} = D_u^s - \frac{3s_{uv}^2}{4}\phi_{uv}\phi_{uv}^T \preceq D_u^s) \\ = & \frac{1}{4}(\phi_{uv}^s)^T (D_u^s)^\dagger \phi_{uv}^s \geqslant \frac{\gamma \cdot k}{4n} \qquad ((\phi_{uv}^s)^T (D_u^s)^\dagger \phi_{uv}^s > \frac{\gamma \cdot k}{n}).\end{aligned}$$

After iteration t_{uv}, we only downscale the edge weights of other edges. Note that for any s'' that is pointwise smaller than s', we have $D_u^{s''} \preceq D_u^{s'}$, and thus the leverage score of (u, v) w.r.t. u can

only increase afterwards. Therefore we have $(\phi_{uv}^s)^T(D_u^s)^\dagger\phi_{uv}^s \geqslant \frac{\gamma\cdot k}{4n}$ in the end, as desired. □

As a consequence of Lemma 4.7.2, the total number of edges that have been downscaled during `whack-a-mole-I` is small.

Corollary 4.7.3 (of Lemma 4.7.2)**.** *Suppose the initial scaling s^0 satisfies that $s_{uv}^0 \in \{0,1\}$ for all $(u,v) \in E$. Then after `whack-a-mole-I` terminates, the number of edges $(u,v) \in E$ with $s_{uv} \in (0,1)$ is at most $\frac{4n\cdot\mathrm{rank}(D^s)}{\gamma\cdot k}$.*

Proof. Note that

$$\begin{aligned}
\sum_{(u,v)\in E}\left((\phi_{uv}^s)^T(D_u^s)^\dagger\phi_{uv}^s + (\phi_{uv}^s)^T(D_v^s)^\dagger(\phi_{uv}^s)\right) &= \sum_{u\in V}\sum_{u\sim v}(\phi_{uv}^s)^T(D_u^s)^\dagger\phi_{uv}^s \\
&= \sum_{u\in V}\sum_{u\sim v}\mathrm{Tr}\left((D_u^s)^\dagger(\phi_{uv}^s)(\phi_{uv}^s)^T\right) \\
&= \sum_{u\in V}\mathrm{Tr}\left((D_u^s)^\dagger\left(\sum_{u\sim v}(\phi_{uv}^s)(\phi_{uv}^s)^T\right)\right) \\
&= \sum_{u\in V}\mathrm{Tr}\left((D_u^s)^\dagger D_u^s\right) \\
&= \sum_{u\in V}\mathrm{rank}\,(D_u^s) = \mathrm{rank}\,(D^s)\,.
\end{aligned}$$

By Lemma 4.7.2, for edges with $s_{uv} \in (0,1)$, at least one of $(\phi_{uv}^s)^T(D_u^s)^\dagger\phi_{uv}^s$, $(\phi_{uv}^s)^T(D_v^s)^\dagger(\phi_{uv}^s)$ is $\geqslant \frac{\gamma\cdot k}{4n}$. Therefore there cannot be more than $\mathrm{rank}\,(D^s)\,/(\frac{\gamma\cdot k}{4n}) = \frac{4n\cdot\mathrm{rank}(D^s)}{\gamma\cdot k}$ such edges. □

Analysis of Algorithm 4.7.2. We first show that all edges that are deleted at Line 13 have been downscaled during `whack-a-mole-I`, and thus by Lemma 4.7.2 have large leverage scores w.r.t. at least one of their endpoints.

Lemma 4.7.4. *The ρ found at Line 12 has the property that for all edges $(u,v) \in E$ such that $\|\phi_{uv}^s\|^2 \leqslant \rho$ at the moment (that is, they will be deleted immediately at the next line), we have $s_{uv} < 1$.*

Proof. First note that before Line 12 we have run `whack-a-mole-I` for $T = n^2(4n^2k\log\alpha + \log(r/\ell))$ iterations. Therefore there must be an edge (u,v) with $s_{uv} \in (0,1)$ whose weight has been halved

at least $4n^2k\log\alpha + \log(r/\ell)$ times. Therefore for such an edge (u,v) we have

$$\begin{aligned}\ell^s = \min_{(u',v')\in E, s_{u'v'}>0} \|\phi^s_{u'v'}\|^2 &\leqslant \|\phi^s_{uv}\|^2 \\ &\leqslant \frac{1}{4^{4n^2k\log\alpha+\log(r/\ell)}} \|\phi_{uv}\|^2 \\ &= \frac{1}{\alpha^{8n^2k}} \cdot \frac{\ell^2}{r^2} \cdot \|\phi_{uv}\|^2 \\ &\leqslant \frac{1}{\alpha^{8n^2k}} \cdot \min_{(u,v)\in E} \|\phi_{uv}\|^2 .\end{aligned}$$

Therefore for any (u,v) whose $\|\phi^s_{uv}\|^2$ falls in $[\ell^s, \ell^s\cdot\alpha^{8n^2k})$, it must be the case that $s_{uv} < 1$. Also, note that the total number of different $\|\phi^s_{uv}\|^2$'s and μ_i's is at most $n^2 + nk \leqslant 2n^2k$. Therefore, as long as $nk > 1$, there must be an interval $(\rho, \rho\cdot\alpha) \subseteq [\ell^s, \ell^s\cdot\alpha^{8n^2k})$ where no $\|\phi^s_{uv}\|^2$ or μ_i resides. Therefore the ρ found at Line 12 must be $< \ell^s\cdot\alpha^{8n^2k}$. So for all edges $(u,v)\in E$ such that $\|\phi^s_{uv}\|^2 \leqslant \rho$ we must have $s_{uv} < 1$. □

We then show that the number of edges deleted at Line 13 of `almost-regular-decomp` is small compared to the rank reduction. To this end let us fix an iteration of the while loop where we go to the "else" branch. Let s denote the scaling returned by `whack-a-mole-I`, and let s' denote the scaling obtained after we delete the small weight edges at Line 13 (but before we reset the weights of other edges at Line 14).

Lemma 4.7.5. *The number of edges $(u,v)\in E$ with $\|\phi^s_{uv}\|^2 \in (0,\rho]$ at Line 13 is at most*

$$\frac{8n}{\gamma\cdot k}\left(\operatorname{rank}(D^s) - \operatorname{rank}\left(D^{s'}\right)\right).$$

Lemma 4.7.6. *We have*

$$\mu_{\max}\left(\left(D^{s'}\right)^{\dagger}\right) \leqslant \frac{1}{4nk\cdot\rho}.$$

Proof. Let $s'_{\min} = \min\{s'_{uv} : s'_{uv} \neq 0\}$. Note that we have

$$(s'_{\min})^2 = \min_{s'_{uv}>0} \frac{\left\|\phi_{uv}^{s'}\right\|^2}{\|\phi_{uv}\|^2} \geqslant \frac{\min_{s'_{uv}>0}\left\|\phi_{uv}^{s'}\right\|^2}{\max_{(u,v)\in E}\|\phi_{uv}\|^2} \geqslant \frac{\rho\cdot\alpha}{r}.$$

Let $F' = \{(u,v) \in E : s'_{uv} > 0\}$. Then we have

$$D^{s'} \succeq (s'_{\min})^2 D^F \succeq \frac{\rho\cdot\alpha}{r}\cdot D^F,$$

and as a result

$$\left(D^{s'}\right)^\dagger \preceq \frac{r}{\rho\cdot\alpha}\cdot\left(D^F\right)^\dagger. \tag{4.30}$$

By Line 3, we have

$$\alpha \geqslant 4nk\cdot r\cdot \max_{F'\subseteq E}\mu_{\max}\left(\left(D^{F'}\right)^\dagger\right) \geqslant 4nk\cdot r\cdot\mu_{\max}\left(\left(D^F\right)^\dagger\right).$$

Plugging this into (4.30) gives

$$\mu_{\max}\left(\left(D^{s'}\right)^\dagger\right) \leqslant \frac{1}{4nk\cdot\rho}.$$

as desired. □

To prove Lemma 4.7.5, we first need to show that the rank of D^s decreases by a certain amount after deleting small weight edges.

Lemma 4.7.7. *Let i be such that $\mu_i \leqslant \rho < \rho\cdot\alpha \leqslant \mu_{i+1}$. Then we have*

$$\operatorname{rank}\left(D^s\right) - \operatorname{rank}\left(D^{s'}\right) \geqslant i.$$

Proof. Let $f_1,\ldots,f_i \in \mathbb{R}^{nk}$ be a set of orthonormal eigenvectors corresponding to $\mu_1,\ldots,\mu_i$. Let $\mathcal{S}\subseteq\mathbb{R}^{nk}$ be the column space of $D^{s'}$. Then it suffices to show that after projecting $f_1,\ldots,f_i$ onto the subspace $\mathcal{S}^\perp$ (the subspace orthogonal to $\mathcal{S}$), they are still linearly independent. To this end,

we first note that for each f_j where $j \leqslant i$, its projection onto $\mathcal{S}$ can be written as

$$\Pi_{\mathcal{S}} f_j = \left(D^{s'}\right)^{\dagger/2} \left(D^{s'}\right)^{1/2} f_j.$$

Using the fact that f_j is an eigenvector of D^s corresponding to eigenvalue $\mu_j \leqslant \rho$, we have

$$\begin{aligned}
\left\| \left(D^{s'}\right)^{1/2} f_j \right\|^2 =& f_j^T D^{s'} f_j \\
\leqslant& f_j^T D^s f_j \qquad (D^{s'} \preceq D^s) \\
=& \mu_j \leqslant \rho.
\end{aligned}$$

Then by Lemma 4.7.6,

$$\begin{aligned}
\left\| \left(D^{s'}\right)^{\dagger/2} \left(D^{s'}\right)^{1/2} f_j \right\|^2 =& f_j^T \left(D^{s'}\right)^{1/2} \left(D^{s'}\right)^{\dagger} \left(D^{s'}\right)^{1/2} f_j \\
\leqslant& \mu_{\max} \left(\left(D^{s'}\right)^{\dagger} \right) f_j^T D^{s'} f_j \\
\leqslant& \frac{1}{4nk \cdot \rho} \cdot \rho = \frac{1}{4nk}.
\end{aligned} \tag{4.31}$$

Let d be the dimension of the subspace $\mathcal{S}^{\perp}$, and let $b_1, \ldots, b_d$ be an orthonormal basis of $\mathcal{S}^{\perp}$. Also, let $b_{d+1}, \ldots, b_{nk} \in \mathbb{R}^{nk}$ be an orthonormal basis of $\mathcal{S}$. Then we can write each f_j as a linear combination of $b_1, \ldots, b_{nk}$. That is, there exists a matrix $C \in \mathbb{R}^{i \times nk}$ such that

$$\begin{pmatrix} f_1^T \\ \vdots \\ f_i^T \end{pmatrix} = C \begin{pmatrix} b_1^T \\ \vdots \\ b_{nk}^T \end{pmatrix}.$$

Since this is an orthogonal transformation, we have that the rows of C are orthonormal, namely $CC^T = I$. If we write the τ^{th} column of C as $c_\tau \in \mathbb{R}^i$, then we have $\sum_{\tau=1}^{nk} c_\tau c_\tau^T = CC^T = I$.

Moreover, we have

$$\sum_{\tau=d+1}^{nk} \|c_\tau\|^2 = \sum_{j=1}^{i} \|\Pi_{\mathcal{S}} f_j\|^2 \leqslant i \cdot \frac{1}{4nk} \leqslant nk \cdot \frac{1}{4nk} = \frac{1}{4},$$

where the first inequality follows from (4.31). Therefore

$$\operatorname{Tr}\left(\sum_{\tau=1}^{d} c_\tau c_\tau^T\right) = \operatorname{Tr}\left(\sum_{\tau=1}^{nk} c_\tau c_\tau^T\right) - \operatorname{Tr}\left(\sum_{\tau=d+1}^{nk} c_\tau c_\tau^T\right) \geqslant i - \frac{1}{4}.$$

On the other hand, we have

$$\sum_{\tau=1}^{d} c_\tau c_\tau^T \preceq \sum_{\tau=1}^{nk} c_\tau c_\tau^T,$$

and thus all eigenvalues of $\sum_{\tau=1}^{d} c_\tau c_\tau^T$ are at most 1. These two imply that $\sum_{\tau=1}^{d} c_\tau c_\tau^T$ must be of full rank i. As a result, $\Pi_{\mathcal{S}^\perp} f_1, \ldots, \Pi_{\mathcal{S}^\perp} f_i$ are linearly independent. □

Proof of Lemma 4.7.5. By Lemma 4.7.7 we know that the rank reduction in D^s is at least i, where i is such that $\mu_i \leqslant \rho < \rho \cdot \alpha \leqslant \mu_{i+1}$. Let $f_1, \ldots, f_p$ be a set of orthonormal eigenvectors corresponding to $\mu_1, \ldots, \mu_p$. By Lemma 4.7.4, we know that for any $(u, v) \in E$ with $\|\phi_{uv}^s\|^2 \in (0, \rho]$, we have $s_{uv} \in (0, 1)$. Let $F \stackrel{\text{def}}{=} \left\{(u, v) \in E : \|\phi_{uv}^s\|^2 \in (0, \rho]\right\}$. As the initial scaling s^0 that we send to `whack-a-mole-I` always has range $\{0, 1\}$, we have by Lemma 4.7.2 that for each $e \in F$, its leverage score is at least $\frac{\gamma \cdot k}{4n}$ w.r.t. at least one of its endpoints.

For each edge (u, v), define two nk-dimensional vectors x_{uv}^u and x_{uv}^v, such that $(x_{uv}^u)_u = \phi_{uv}^s$ and $(x_{uv}^u)_w = 0$ for all $w \neq u$, and $(x_{uv}^v)_v = \phi_{uv}^s$ and $(x_{uv}^v)_w = 0$ for all $w \neq v$. Then for any $e \in F$ we have

$$(x_{uv}^u)^T (D^s)^\dagger x_{uv}^u + (x_{uv}^v)^T (D^s)^\dagger x_{uv}^v \geqslant \frac{\gamma \cdot k}{4n}.$$

Now if we restrict to only the eigenvalues smaller than ρ, we have

$$\begin{aligned}
&(x_{uv}^u)^T \left(\sum_{j=1}^{i} \mu_j f_j f_j^T\right)^\dagger x_{uv}^u + (x_{uv}^v)^T \left(\sum_{j=1}^{i} \mu_j f_j f_j^T\right)^\dagger x_{uv}^v \\
=&(x_{uv}^u)^T \left((D^s)^\dagger - \sum_{j=i+1}^{p} \frac{1}{\mu_j} f_j f_j^T\right) x_{uv}^u + \\
&(x_{uv}^v)^T \left((D^s)^\dagger - \sum_{j=i+1}^{p} \frac{1}{\mu_j} f_j f_j^T\right) x_{uv}^v. \qquad (4.32)
\end{aligned}$$

Notice that for all $e \in F$ we have $\|x_{uv}^u\|^2 = \|x_{uv}^v\|^2 = \|\phi_{uv}^s\|^2 \leqslant \rho$. And thus for any $j \geqslant i+1$,

$$\begin{aligned}
\frac{1}{\mu_j} \langle x_{uv}^u, f_j \rangle^2 \leqslant \frac{\rho}{\mu_j} \leqslant \frac{1}{\alpha}, \\
\frac{1}{\mu_j} \langle x_{uv}^v, f_j \rangle^2 \leqslant \frac{\rho}{\mu_j} \leqslant \frac{1}{\alpha},
\end{aligned}$$

where both second inequalities follow from $\mu_j \geqslant \rho \cdot \alpha$ for $j \geqslant i+1$. Now by Line 3, $\alpha \geqslant 8n^4k$. Therefore combining this with (4.32) we have

$$\begin{aligned}
&\sum_{(u,v)\in F} (x_{uv}^u)^T \left(\sum_{j=1}^{i} \mu_j f_j f_j^T\right)^\dagger x_{uv}^u + (x_{uv}^v)^T \left(\sum_{j=1}^{i} \mu_j f_j f_j^T\right)^\dagger x_{uv}^v \\
\geqslant &\sum_{(u,v)\in F} (x_{uv}^u)^T (D^s)^\dagger x_{uv}^u + (x_{uv}^v)^T (D^s)^\dagger x_{uv}^v - 2nk \cdot \frac{1}{8n^4k} \\
\geqslant &|F| \cdot \left(\frac{\gamma \cdot k}{4n} - \frac{1}{4n^3}\right) \geqslant |F| \cdot \frac{\gamma \cdot k}{8n}.
\end{aligned}$$

On the other hand, we have

$$
\begin{aligned}
&\sum_{(u,v)\in F} (x_{uv}^u)^T \left(\sum_{j=1}^{i} \mu_j f_j f_j^T\right)^{\dagger} x_{uv}^u + (x_{uv}^v)^T \left(\sum_{j=1}^{i} \mu_j f_j f_j^T\right)^{\dagger} x_{uv}^v \\
=&\mathrm{Tr}\left(\left(\sum_{j=1}^{i} \mu_j f_j f_j^T\right)^{\dagger/2} \left(\sum_{(u,v)\in F} x_{uv}^u \left(x_{uv}^u\right)^T + x_{uv}^v \left(x_{uv}^v\right)^T\right) \left(\sum_{j=1}^{i} \mu_j f_j f_j^T\right)^{\dagger/2}\right) \\
\leqslant&\mathrm{Tr}\left(\left(\sum_{j=1}^{i} \mu_j f_j f_j^T\right)^{\dagger/2} \left(\sum_{(u,v)\in E} x_{uv}^u \left(x_{uv}^u\right)^T + x_{uv}^v \left(x_{uv}^v\right)^T\right) \left(\sum_{j=1}^{i} \mu_j f_j f_j^T\right)^{\dagger/2}\right) \\
=&\mathrm{Tr}\left(\left(\sum_{j=1}^{i} \mu_j f_j f_j^T\right)^{\dagger/2} D^s \left(\sum_{j=1}^{i} \mu_j f_j f_j^T\right)^{\dagger/2}\right) \\
=&\mathrm{Tr}\left(\left(\sum_{j=1}^{i} \mu_j f_j f_j^T\right)^{\dagger/2} \left(\sum_{j=1}^{p} \mu_j f_j f_j^T\right) \left(\sum_{j=1}^{i} \mu_j f_j f_j^T\right)^{\dagger/2}\right) \qquad \text{(spectral decomposition)} \\
=&\mathrm{rank}\left(\sum_{j=1}^{i} \mu_j f_j f_j^T\right) \leqslant i.
\end{aligned}
$$

Combining the above two inequalities we have

$$
|F| \leqslant \frac{8ni}{\gamma \cdot k} \leqslant \frac{8n}{\gamma \cdot k}\left(\mathrm{rank}\left(D^s\right) - \mathrm{rank}\left(D^{s'}\right)\right),
$$

as desired. □

Proof of Theorem 4.7.1. Since in each while loop iteration we zero out the weight of at least one edge, the while loop must terminate in $O(n^2)$ iterations, and thus the algorithm terminates in finite time. Also by the termination condition of the while loop, the resulting graph G^s is γ-almost regular. Finally, Corollary 4.7.3 and Lemma 4.7.5 give the desired bounds on the number of downscaled/deleted edges. □

4.8. Almost regular expander decomposition

In this section, we show that every matrix-weighted graph that is sufficiently dense can be made into an almost regular "expander" graph, by downscaling a small number of edges. This can be seen as a matrix-weighted analog of the celebrated expander decomposition of scalar-weighted graphs [GR99, KVV04, ST04].

We first give the definition of matrix-weighted expanders. Consider a $k \times k$ matrix-weighted graph $G = (V, E)$ with edge weights $\phi_{uv}\phi_{uv}^T$. Let $\lambda_1 \leqslant \ldots \lambda_r$, where $r = \operatorname{rank}(D)$, be the nontrivial eigenvalues (Definition 4.5.20) of its normalized Laplacian $N = D^{\dagger/2} L D^{\dagger/2}$. Let $f_1, \ldots, f_r$ be a corresponding set of orthonormal, nontrivial eigenvectors, which by definition of nontriviality satisfies

$$(f_i)_u \perp \ker(D_u), \ \forall i \in [r], u \in V.$$

Definition 4.8.1 (Almost regular matrix-weighted expanders)**.** For $\gamma \geqslant 1$, $\zeta \in (0, 1)$, and $\psi \geqslant 1$, we say G is a (γ, ζ, ψ)-almost regular expander if

1. (γ-almost regularity) For every vertex u and every incident edge $(u, v) \in E$, we have

$$\phi_{uv}^T D_u^\dagger \phi_{uv} \leqslant \frac{\gamma \cdot k}{n}.$$

2. ((ζ, ψ)-expander) for every edge $(u, v) \in E$ we have

$$\left(D^{\dagger/2} b_{uv}\right)^T \left(\sum_{\lambda_i \in (0, \zeta]} \frac{1}{\lambda_i} f_i f_i^T \right) \left(D^{\dagger/2} b_{uv}\right) \leqslant \frac{\psi \cdot k^2}{n^2} \tag{4.33}$$

Notations for rescaled graphs. In describing our main result in this section, we will use similar notations for rescaled graphs as in Section 4.7. Namely, for a $k \times k$ matrix-weighted graph $G = (V, E)$ with edge weights $\phi_{uv}\phi_{uv}^T$'s and a scaling $s : E \to [0, 1]$, we will write G^s to denote the graph obtained from G by rescaling each edge (u, v)'s weight to $(s_{uv}\phi_{uv})(s_{uv}\phi_{uv})^T$. For ease of presentation we will use the superscript s when dealing with vectors and matrices associated with G^s. For

instance, we write $\phi^s_{uv} = s_{uv}\phi_{uv}$, $D^s = \sum_{u\sim v} s^2_{uv}(e_{u\leftarrow v}e^T_{u\leftarrow v}+e_{v\leftarrow u}e^T_{v\leftarrow u})$, and $L^s = \sum_{u\sim v} s^2_{uv} b_{uv}b^T_{uv}$. Analogously, for a subset of edges $F \subseteq E$, we also use the superscript F when dealing with matrices associated with the induced subgraph $G[F]$, and thus, for instance, $D^F = \sum_{(u,v)\in F}(e_{u\leftarrow v}e^T_{u\leftarrow v} + e_{v\leftarrow u}e^T_{v\leftarrow u})$.

We will write $\lambda^s_1 \leqslant \ldots \leqslant \lambda^s_{\mathrm{rank}(D^s)}$ to denote the nontrivial eigenvalues of the normalized Laplacian $N^s = (D^s)^{\dagger/2}L^s(D^s)^{\dagger/2}$ of G^s, and write $f^s_1, \ldots, f^s_{\mathrm{rank}(D^s)}$ to denote a corresponding set of orthonormal, nontrivial eigenvectors. We then define a function $H^s_\zeta : E \to \mathbb{R}$ by

$$H^s_\zeta(u,v) \stackrel{\mathrm{def}}{=} \left((D^s)^{\dagger/2} b^s_{uv}\right)^T \left(\sum_{\lambda^s_i \in (0,\zeta]} \frac{1}{\lambda^s_i} f^s_i (f^s_i)^T\right) \left((D^s)^{\dagger/2} b^s_{uv}\right).$$

We also define another function $R^s : E \to \mathbb{R}$ by

$$\begin{aligned} R^s(u,v) \stackrel{\mathrm{def}}{=} &(e^s_{u\leftarrow v})^T (D^s)^\dagger e^s_{u\leftarrow v} + (e^s_{v\leftarrow u})^T (D^s)^\dagger e^s_{v\leftarrow u} \\ =&(\phi^s_{uv})^T (D^s_u)^\dagger \phi^s_{uv} + (\phi^s_{uv})^T (D^s_v)^\dagger \phi^s_{uv}. \end{aligned}$$

Then, for our goal, it suffices to find a scaling s such that for all $(u,v) \in E$ we have

$$R^s(u,v) \leqslant \frac{\gamma \cdot k}{n} \qquad \text{and} \qquad H^s_\zeta(u,v) \leqslant \frac{\psi \cdot k^2}{n^2}.$$

Our main result here is a deterministic algorithm for finding a large almost regular expander subgraph within any given graph that is sufficiently dense.

Theorem 4.8.2. *There is a deterministic algorithm* `expander-decomp` *that, given any $k \times k$ matrix-weighted graph $G = (V, E)$ with edge weights $\phi_{uv}\phi^T_{uv}$'s, any $\gamma \geqslant 1$, $\zeta \in (0,1)$, and*

$$\psi \geqslant \frac{1024\gamma^2}{(1-\zeta)^2},$$

outputs a scaling $s : E \to [0,1]$ such that

1. *The rescaled graph G^s is a (γ, ζ, ψ)-almost regular expander.*

2. The number of edges $(u,v) \in E$ with $s_{uv} < 1$ is at most

$$\frac{1000n^2}{\gamma} + \frac{100000n^2\gamma^2k^2}{\psi(1-\zeta)^2}.$$

The algorithm terminates in finite time.

Note that similar to Section 4.7, our goal is to prove the existence of such a scaling for any given weights $\phi_{uv}\phi_{uv}^T$'s that may potentially have *infinite* precision, so we only focus on designing an algorithm that terminates in finite time, as opposed to giving an explicit bound on its running time.

We give the pseudocodes of our algorithm `expander-decomp` in Algorithm 4.8.2 and its subroutine `whack-a-mole-II` in Algorithm 4.8.1. At a high level, in `expander-decomp`, we first try to eliminate edges with large $H_\zeta^s(u,v)$ values by iteratively halving the weight of any such edge for some large number of iterations, while simultaneously maintaining the almost regularity of the graph, by also iteratively halving the weight of any edge with large $R^s(u,v)$ value. This iterative process is essentially achieved by `whack-a-mole-II`. Then after `whack-a-mole-II` terminates, either (i) all edges have small $H_\zeta^s(u,v)$ and $R^s(u,v)$ values, in which case we are done, or (ii) we can strictly increase[10] the number of nontrivial (Definition 4.5.20) zero eigenvalues of the normalized Laplacian by removing a small number of edges that have low weights after `whack-a-mole-II`. Since we maintain the almost regularity of the graph throughout, by Theorem 4.6.2, the total number of nontrivial zero eigenvalues is small, therefore we will only repeat (ii) for a limited number of times.

Since we want to ensure that the number of rescaled edges is small in `whack-a-mole-II`, we maintain the invariant that any edge that has been downscaled satisfies that either $H_\zeta^s(u,v)$ is large or $R^s(u,v)$ is large, by iteratively *doubling* the weight of any edge with both values small. In order to make sure that there is consistent progress in this process, we set up two thresholds $\psi_1 < \psi_2$ for $H_\zeta^s(u,v)$ and also two thresholds $\gamma_1 < \gamma_2$ for $R^s(u,v)$. Then we halve the weight of an edge (u,v) if either $H_\zeta^s(u,v) > \frac{\psi_2 k^2}{n^2}$ or $R^s(u,v) > \frac{\gamma_2 k}{n}$, but only double its weight if *both* $H_\zeta^s(u,v) < \frac{\psi_1 k^2}{n^2}$ *and* $R^s(u,v) < \frac{\gamma_1 k}{n}$. To measure the progress in this elaborate process, we will need to use some global quantities as our potential functions. Specifically, we will design our potential functions based on

[10]This statement is a slight simplification of our actual analysis, but helps with getting intuition.

certain variants of determinant, which we define below.

Definition 4.8.3. For a positive semidefinite matrix $M \in \mathbb{R}^{N \times N}$ with eigenvalues

$$0 = \lambda_1 = \cdots = \lambda_z < \lambda_{z+1} \leqslant \ldots \leqslant \lambda_N,$$

define

$$\det_+(M) \stackrel{\text{def}}{=} \prod_{\lambda_i > 0} \lambda_i.$$

For an integer ℓ, define

$$\det_\ell(M) \stackrel{\text{def}}{=} \prod_{i=z+1}^{z+\ell} \lambda_i.$$

We will need the following variational characterization of $\det_\ell$, whose proof is deferred to Appendix B.4.

Lemma 4.8.4. *Let $f_1, \ldots, f_\ell$ be a set of orthonormal eigenvectors corresponding to the ℓ smallest nonzero eigenvalues of N, and let $F = \begin{pmatrix} f_1 & \ldots & f_\ell \end{pmatrix} \in \mathbb{R}^{nk \times \ell}$. Let $\mathcal{X} \subseteq \mathbb{R}^{nk \times \ell}$ be the set of all full-rank $nk \times \ell$ matrices whose columns are in the range of N. Then*

$$\det_\ell(N) = \min_{X \in \mathcal{X}} \frac{\det\left(X^T L X\right)}{\det\left(X^T D X\right)},$$

and a minimizer can be obtained by letting $X^* = D^{\dagger/2} F$.

The following matrix determinant lemma will be helpful in our analysis of the potential functions.

Lemma 4.8.5 ([Har98])**.** *For an invertible square matrix $M \in \mathbb{R}^{d \times d}$ and vectors $x, y \in \mathbb{R}^d$, we have*

$$\det(M + xy^T) = (1 + y^T M^{-1} x) \det(M).$$

Algorithm 4.8.1 `whack-a-mole-II`$(G, \gamma_1, \gamma_2, \zeta, \psi_1, \psi_2, \alpha, s^0)$

Input: G: a $k \times k$ matrix-weighted graph $G = (V, E)$ with edge weights $\phi_{uv}\phi_{uv}^T$.
$\gamma_1 < \gamma_2, \zeta, \psi_1 < \psi_2$: parameters for almost regularity and expander.
$\alpha > 0$: a gap parameter.
s^0: an initial scaling, mapping from $E \to [0, 1]$, s.t. G^{s_0} is $\gamma_1/2$-almost regular.

Output: s: a scaling, mapping from $E \to [0, 1]$.
ρ: a threshold.

Procedure:

Let $s \leftarrow s^0$.
while true **do**
  Compute the eigenvalues $\lambda_1 \leqslant \ldots \leqslant \lambda_{nk}$ of $(D^s)^{\dagger/2} L^s (D^s)^{\dagger/2}$.
  Compute the eigenvalues $\mu_1 \leqslant \ldots \leqslant \mu_{nk}$ of D^s.
  if no λ_i is in the range $(0, \zeta]$ **then**
    Return s and ρ, and halt.
  Let $\lambda_{\min} = \min_{\lambda_i \in (0,\zeta]} \lambda_i$.
  Find the smallest $\rho \geqslant \lambda_{\min}$ s.t. all $(s_{uv}/s^0_{uv})^2$, μ_i, and λ_i fall in $\mathbb{R} \setminus (\rho, \rho \cdot \alpha)$.
  if $\rho < 1$ **then**
    Return s and ρ, and halt.
  else if $\nexists (u, v) \in E$ with $H^s_\zeta(u, v) > \frac{\psi_2 \cdot k^2}{n^2}$ **then**
    Return s and ρ, and halt.
  else
    Let $s_{uv} \leftarrow s_{uv}/2$ for an arbitrary edge (u, v) with $H^s_\zeta(u, v) > \frac{\psi_2 \cdot k^2}{n^2}$.
    while $\exists (u, v) \in E$ with $R^s(u, v) > \frac{\gamma_2 \cdot k}{n}$ **do**
      Let $s_{uv} \leftarrow s_{uv}/2$ for an arbitrary such edge (u, v).
    while $\exists (u, v) : s_{uv} < s^0_{uv}$ satisfying both $R^s(u, v) < \frac{\gamma_1 \cdot k}{n}$ and $H^s_\zeta(u, v) < \frac{\psi_1 \cdot k^2}{n^2}$ **do**
      Let $s_{uv} \leftarrow s_{uv} \cdot 2$ for an arbitrary such edge (u, v).

Algorithm 4.8.2 `expander-decomp`(G, γ, ζ, ψ)

Input: G: a $k \times k$ matrix-weighted graph $G = (V, E)$ with edge weights $\phi_{uv}\phi_{uv}^T$.
γ, ζ, ψ: parameters for almost regularity and expander.

Output: s: a scaling, mapping from $E \to [0, 1]$.

$r_\phi \leftarrow \max_{(u,v)\in E} \|\phi_{uv}\|^2$. **Procedure:**

Let $\alpha_1 \leftarrow \max_{F\subseteq E} \lambda_{\max}\left(\left(D^F\right)^\dagger\right)$.
Let $\alpha_2 \leftarrow \max_{F\subseteq E} \lambda_{\max}\left(\left(D^{\dagger/2}L^F D^{\dagger/2}\right)^\dagger\right)$.
Initially, let $s_e \leftarrow 1$ for all $e \in E$.
while $\operatorname{rank}(D^s) > 0$ **do**
    Let $G' = (V, E')$ where $E' = \{e : s_e > 0\}$.
    Let $s' \leftarrow$ `almost-regular-decomp`$(G', \gamma/32)$, and then $s_e \leftarrow s'_e$ for all $e \in E'$.
    $s'_{\min} \leftarrow \min_{s'_{uv}>0} s'_{uv}$.
    Let $\alpha \leftarrow \max\left\{8n^4k, 128nk^3\gamma \cdot r_\phi \cdot \alpha_1/(s'_{\min})^2, 128nk^3\gamma \cdot \alpha_2/(s'_{\min})^2\right\}$.
    $(s, \rho) \leftarrow$ `whack-a-mole-II`$(G, \gamma/16, \gamma, \zeta, \psi/1024, \psi, \alpha, s)$.
    if $\forall (u, v) \in E$ we have $R^s(u, v) \leqslant \frac{\gamma\cdot k}{n}$ and $H^s_\zeta(u, v) \leqslant \frac{\psi\cdot k^2}{n^2}$ **then**
        Return s and halt.
    else
        Set $s_{uv} \leftarrow 0$ for all $(u, v) \in E$ with $(s_{uv}/s'_{uv})^2 \leqslant \rho$.
        Set $s_{uv} \leftarrow 1$ for all $(u, v) \in E$ with $s_{uv} > 0$.
return s.

4.8.1. Analysis of Algorithm 4.8.1

For a fixed scaling $s : E \to [0, 1]$, let ℓ be the number of eigenvalues of the normalized Laplacian N^s of G^s that are in the range $(0, \zeta]$. Let $\ell_0 = \left\lceil \frac{\gamma_2 k^2}{(1-\zeta)^2} \right\rceil$, which, by Theorem 4.6.2, is an upper bound on ℓ when the graph G^s is γ_2-almost regular. Then we consider the following potential function:

$$\Upsilon(s) \stackrel{\text{def}}{=} \begin{cases} \zeta^{\ell_0-\ell} \det_\ell(N^s) & \ell \leqslant \ell_0 \\ \det_{\ell_0}(N^s) & \ell > \ell_0. \end{cases} \tag{4.34}$$

Thus, $\Upsilon(s)$ is always a product of exactly ℓ_0 numbers between $(0, \zeta]$. It is not hard to see the following alternative form of $\Upsilon(s)$, which will be helpful for analyzing it.

Proposition 4.8.6. *Let $\lambda_1, \ldots, \lambda_{\ell_0}$ be the smallest ℓ_0 nonzero eigenvalues of N^s. Then*

$$\Upsilon(s) = \prod_{i=1}^{\ell_0} \min\{\lambda_i, \zeta\}.$$

Corollary 4.8.7 (of Proposition 4.8.6)**.** *For any $\ell' \leqslant \ell_0$, we have $\Upsilon(s) \leqslant \det_{\ell'}(N^s)\zeta^{\ell_0 - \ell'}$.*

Since we always maintain the γ_2-regularity, we will also need to consider the determinant of the degree matrix, $\det_+(D)$, as a potential function. We show that starting from an almost-regular graph, $\det_+(D)$ can only decrease at a limited speed when edges are downscaled. The proof of the lemma below is deferred to Appendix B.4.

Lemma 4.8.8. *Let $G = (V, E)$ be a γ-almost regular graph. Let $s : E \to (0, 1]$ be a strictly positive scaling, and let $S \stackrel{\text{def}}{=} \prod_{e \in E} \frac{1}{s_e}$. Then we have*

$$\det_+(D^s) \geqslant \left(1 - \frac{2\gamma k}{n}\right)^{2 \log S} \det_+(D).$$

As a result of the above lemma, we then show that the while loop at Lines 15-16 of the algorithm `whack-a-mole-II` will terminate after a bounded number of iterations. The proof of the following lemma is also deferred to Appendix B.4.

Lemma 4.8.9. *Suppose the input to `whack-a-mole-II` satisfies that G^{s_0} is $\gamma_1/2$-regular and $\gamma_2 \geqslant 16\gamma_1$. Consider the first t iterations of the outermost while loop. Then the total number of iterations executed so far by the inner while loop at Lines 15-16 is at most $2t$.*

We now prove our key lemma, which shows that our potential Υ will decrease at least as fast as a geometric series with rate bounded away from 1. As a result, the smallest nonzero eigenvalue of the normalized Laplacian must also decrease at a steady rate. This implies that eventually we will be able to find a $\rho < 1$ at Line 8, and therefore terminate the outermost while loop in finite time.

Lemma 4.8.10. *Suppose the input to `whack-a-mole-II` satisfies that G^{s_0} is $\gamma_1/2$-regular, $\gamma_2 \geqslant$*

$16\gamma_1$, $\psi_2 \geqslant 1024\psi_1$, *and*

$$\psi_2 \geqslant \frac{1024\gamma_2^2}{(1-\zeta)^2}.$$

Consider the first t iterations of the outermost while loop, and let s^1 be the scaling obtained at the end of the t^{th} iteration. Then

$$\frac{\Upsilon(s^1)}{\Upsilon(s^0)} \leqslant \left(1 - \frac{\psi_2 k^2}{16n^2}\right)^t.$$

Proof. We first show that the potential reduces by a certain amount at Line 14.

Claim 4.8.11. *Let the scalings before and after one execution of Line 14 be s and s' respectively. Then we have*

$$\frac{\Upsilon(s')}{\Upsilon(s)} \leqslant 1 - \frac{\psi_2 k^2}{4n^2}.$$

Proof of Claim 4.8.11. Let ℓ be the number of eigenvalues of N^s that are between $(0, \zeta]$. Since G^s is γ_2-almost regular, we have $\ell \leqslant \ell_0$ by Theorem 4.6.2. Thus $\Upsilon(s) = \det_\ell(N^s)\zeta^{\ell_0-\ell}$. Also, by Corollary 4.8.7, we have $\Upsilon(s') \leqslant \det_\ell(N^{s'})\zeta^{\ell_0-\ell}$. Therefore, it suffices to show that

$$\frac{\det_\ell(N^{s'})}{\det_\ell(N^s)} \leqslant 1 - \frac{\psi_2 k^2}{4n^2}. \tag{4.35}$$

We then do so by considering the variational characterization $\det_\ell$ from Lemma 4.8.4. Let $X = (D^s)^{\dagger/2}F$ and $X' = (D^{s'})^{\dagger/2}F'$ be the optimal matrix that minimizes the variational characterization of $\det_\ell(N^s)$ and $\det_\ell(N^{s'})$ respectively. Here F and F' are both $nk \times \ell$ matrices whose columns are bottom nonzero eigenvectors of N^s and $N^{s'}$ respectively.

It then suffices to show that the fraction in the characterization reduces by much after the execution of Line 14, even if we do not switch from X to X', since switching to the latter can only decrease

the potential function. Namely, since by the optimality of X' we have

$$\frac{\det\left(X^T L^{s'} X\right)}{\det\left(X^T D^{s'} X\right)} \geqslant \frac{\det\left((X')^T L^{s'}(X')\right)}{\det\left((X')^T D^{s'}(X')\right)},$$

it suffices to show

$$\left(\frac{\det\left(X^T L^{s'} X\right)}{\det\left(X^T D^{s'} X\right)}\right) \Big/ \left(\frac{\det\left(X^T L^{s} X\right)}{\det\left(X^T D^{s} X\right)}\right) \leqslant 1 - \frac{\psi_2 k^2}{4n^2}. \tag{4.36}$$

For the numerator we have by matrix determinant lemma

$$\begin{aligned}
\frac{\det\left(X^T L^{s'} X\right)}{\det\left(X^T L^{s} X\right)} =& 1 - \frac{3}{4}\cdot (b^s_{uv})^T X \left(X^T L^s X\right)^{-1} X^T (b^s_{uv}) \\
=& 1 - \frac{3}{4}\cdot (b^s_{uv})^T \left(D^s\right)^{\dagger/2} F \left(F^T \left(D^s\right)^{\dagger/2} L^s \left(D^s\right)^{\dagger/2} F\right)^{-1} F^T \left(D^s\right)^{\dagger/2} b^s_{uv} \\
=& 1 - \frac{3}{4}\cdot H^s_\zeta(u,v) \leqslant 1 - \frac{3\psi_2 k^2}{4n^2},
\end{aligned} \tag{4.37}$$

where the last equality follows from that F's columns are eigenvectors corresponding to all eigenvalues of N^s that are between $(0, \zeta]$. For the denominator, we have, again by matrix determinant lemma,

$$\begin{aligned}
&\frac{\det\left(X^T D^{s'} X\right)}{\det\left(X^T D^{s} X\right)} \\
\geqslant& 1 - \frac{3}{4}(e^s_{u\leftarrow v})^T X (X^T D^s X)^{-1} X^T e^s_{u\leftarrow v} - \frac{3}{4}(e^s_{v\leftarrow u})^T X (X^T D^s X)^{-1} X^T e^s_{v\leftarrow u} \\
=& 1 - \frac{3}{4}(e^s_{u\leftarrow v})^T X X^T e^s_{u\leftarrow v} - \frac{3}{4}(e^s_{v\leftarrow u})^T X X^T e^s_{v\leftarrow u} \qquad (\text{as } X^T D^s X = F^T F = I) \\
=& 1 - \frac{3}{4}(e^s_{u\leftarrow v})^T (D^s)^{\dagger/2} F F^T (D^s)^{\dagger/2} (e^s_{u\leftarrow v}) - \frac{3}{4}(e^s_{v\leftarrow u})^T (D^s)^{\dagger/2} F F^T (D^s)^{\dagger/2} (e^s_{v\leftarrow u}) \\
=& 1 - \frac{3}{4}\left\|F^T(u)(D^s_u)^{\dagger/2}\phi_{uv}\right\|^2 - \frac{3}{4}\left\|F^T(v)(D^s_v)^{\dagger/2}\phi_{uv}\right\|^2,
\end{aligned}$$

where in the last equality we let $F(u)$ be the u^{th} row block of F. Since G^s is γ_2-almost regular, we have $\left\|(D^s_u)^{\dagger/2}\phi_{uv}\right\|^2 \leqslant \frac{\gamma_2 \cdot k}{n}$ and $\left\|(D^s_v)^{\dagger/2}\phi_{uv}\right\|^2 \leqslant \frac{\gamma_2 \cdot k}{n}$. Note that by Lemma 4.6.4, we have

$\lambda_{\max}(F(u)F(u)^T) \leqslant \frac{\gamma_2 k}{(1-\zeta)^2 n}$, and $\lambda_{\max}(F(v)F(v)^T) \leqslant \frac{\gamma_2 k}{(1-\zeta)^2 n}$, and therefore

$$\left\|F^T(u)(D_u^s)^{\dagger/2}\phi_{uv}\right\|^2 \leqslant \frac{\gamma_2^2 \cdot k^2}{(1-\zeta)^2 n^2}$$
$$\left\|F^T(v)(D_v^s)^{\dagger/2}\phi_{uv}\right\|^2 \leqslant \frac{\gamma_2^2 \cdot k^2}{(1-\zeta)^2 n^2}.$$

This give us

$$\frac{\det\left(X^T D^{s'} X\right)}{\det\left(X^T D^s X\right)} \geqslant 1 - \frac{3\gamma_2^2 \cdot k^2}{2(1-\zeta)^2 n^2}. \tag{4.38}$$

(4.37),(4.38) coupled with $\psi_2 \geqslant \frac{1024\gamma_2^2}{(1-\zeta)^2}$ imply (4.36), which finishes the proof of the claim. □

We next show that during the first while loop at Lines 15-16, the potential function cannot increase much.

Claim 4.8.12. *Consider a fixed iteration of the outermost while loop. Suppose in this iteration of the outer while loop, the total number of iterations executed by the first inner while loop at Lines 15-16 is t_1. Let s, s' be the scalings before and after the while loop respectively. Then*

$$\frac{\Upsilon(s')}{\Upsilon(s)} \leqslant \left(1 - \frac{96\gamma_2^2 k^2}{(1-\zeta)^2 n^2}\right)^{-t_1}.$$

Proof of Claim 4.8.12. We know that before Line 14, the graph is γ_2-almost regular. Then at Line 14 we halve the scale of a single edge, so the graph G^s is $4\gamma_2$-almost regular afterwards. We also know that after the while loop terminates at Line 16, the graph $G^{s'}$ is γ_2-almost regular, by the termination condition of the while loop. Now consider the following process for obtaining s' from s.

1. While $s \neq s'$:
   (a) For each $(u, v) \in E$ such that $s_{uv} > s'_{uv}$, let $s_{uv} \leftarrow s_{uv}/2$.

We first argue, by induction, that at the end of each iteration of the while loop of the above process, G^s is $4\gamma_2$-almost regular. As noted above, initially, the graph G^s is $4\gamma_2$-almost regular. For the induction step, consider a fixed iteration, and let F be the edges (u, v) for which $s_{uv} > s'_{uv}$. Since

we decrease the weights of all edges in F by a same factor, from the point view of leverage scores, it is equivalent to increase the weights of all other edges by a same multiple. Therefore the leverage scores of edges in F can only decrease, and thus can be at most $\frac{4\gamma_2 k}{n}$ after this iteration. As for edges (u,v) not in F, they satisfy $s_{uv} = s'_{uv}$. We know that in $G^{s'}$ their leverage scores are at most $\frac{4\gamma_2 k}{n}$, as $G^{s'}$ is γ_2-almost regular. Since G^s's weights always dominate those of G^s, their leverage scores in G^s can only be smaller, and thus at most $\frac{4\gamma_2 k}{n}$ as well.

We then argue that at any point of the process, the graph is $16\gamma_2$-almost regular. This follows by noting that at any point of the algorithm, G^s's edge weights are within a factor 4 of those at the end of previous iteration, and those at the end of the current iteration.

We now show that in this process, each time we let $s_{uv} \leftarrow s_{uv}/2$, the potential function increases by at most $(1 - \frac{400\gamma_2^2 k^2}{(1-\zeta)^2 n^2})^{-1}$, which implies the statement of this claim. Let q, q' be the scalings before and after one execution of $s_{uv} \leftarrow s_{uv}/2$. Let ℓ be the number of nonzero eigenvalues of N^q that are between $(0, \zeta]$, and let $\ell_1 = \min(\ell, \ell_0)$. Then we have $\Upsilon(q) = \det_{\ell_1}(N^q)\zeta^{\ell_0 - \ell_1}$, and, by Corollary 4.8.7, $\Upsilon(q') \leqslant \det_{\ell_1}(N^{q'})\zeta^{\ell_0-\ell_1}$. Thus it suffices to show

$$\frac{\det_{\ell_1}(N^{q'})}{\det_{\ell_1}(N^q)} \leqslant \left(1 - \frac{400\gamma_2^2 k^2}{(1-\zeta)^2 n^2}\right)^{-1}. \tag{4.39}$$

As in our proof of Claim 4.8.11, we again consider the variational characterization in Lemma 4.8.4. Let $X = (D^q)^{\dagger/2} F$ where $F \in \mathbb{R}^{nk \times \ell_1}$'s columns are nonzero bottom eigenvectors of N^q. The numerator of the characterization can only decrease. For the denominator, we have by matrix determinant lemma

$$\begin{aligned}
&\frac{\det\left(X^T D^{q'} X\right)}{\det\left(X^T D^q X\right)} \\
\geqslant &1 - \frac{3}{4}(e^q_{u\leftarrow v})^T X (X^T D^q X)^{-1} X^T e^q_{u\leftarrow v} - \frac{3}{4}(e^q_{v\leftarrow u})^T X (X^T D^q X)^{-1} X^T e^q_{v\leftarrow u} \\
= &1 - \frac{3}{4}(e^q_{u\leftarrow v})^T X X^T e^q_{u\leftarrow v} - \frac{3}{4}(e^q_{v\leftarrow u})^T X X^T e^q_{v\leftarrow u} \qquad (\text{as } X^T D^q X = F^T F = I) \\
= &1 - \frac{3}{4}(e^q_{u\leftarrow v})^T (D^q)^{\dagger/2} F F^T (D^q)^{\dagger/2} (e^q_{u\leftarrow v}) - \frac{3}{4}(e^q_{v\leftarrow u})^T (D^q)^{\dagger/2} F F^T (D^q)^{\dagger/2} (e^q_{v\leftarrow u}) \\
= &1 - \frac{3}{4}\left\|F^T(u)(D^q_u)^{\dagger/2}\phi_{uv}\right\|^2 - \frac{3}{4}\left\|F^T(v)(D^q_v)^{\dagger/2}\phi_{uv}\right\|^2,
\end{aligned}$$

where in the last equality we let $F(u)$ be the u^{th} row block of F. Since G^q is $16\gamma_2$-almost regular, we have $\left\|(D_u^q)^{\dagger/2}\phi_{uv}\right\|^2 \leqslant \frac{16\gamma_2\cdot k}{n}$ and $\left\|(D_v^q)^{\dagger/2}\phi_{uv}\right\|^2 \leqslant \frac{16\gamma_2\cdot k}{n}$. By Lemma 4.6.4, we have $\lambda_{\max}(F(u)F(u)^T) \leqslant \frac{16\gamma_2 k}{(1-\zeta)^2 n}$, and $\lambda_{\max}(F(v)F(v)^T) \leqslant \frac{16\gamma_2 k}{(1-\zeta)^2 n}$, and therefore

$$\left\|F^T(u)(D_u^q)^{\dagger/2}\phi_{uv}\right\|^2 \leqslant \frac{256\gamma_2^2\cdot k^2}{(1-\zeta)^2 n^2}$$
$$\left\|F^T(v)(D_v^q)^{\dagger/2}\phi_{uv}\right\|^2 \leqslant \frac{256\gamma_2^2\cdot k^2}{(1-\zeta)^2 n^2}.$$

This give us

$$\frac{\det\left(X^T D^{q'} X\right)}{\det\left(X^T D^q X\right)} \geqslant 1 - \frac{400\gamma_2^2\cdot k^2}{(1-\zeta)^2 n^2}. \tag{4.40}$$

This then implies (4.39) and finishes the proof of the claim. □

We next show that during the second while loop at Lines 17-18, the potential function cannot increase much either.

Claim 4.8.13. *Consider a fixed iteration of the outermost while loop. Suppose in this iteration of the outer while loop, the total number of iterations executed by the first inner while loop at Lines 17-18 is t_2. Let s, s' be the scalings before and after the while loop respectively. Then*

$$\frac{\Upsilon(s')}{\Upsilon(s)} \leqslant \left(1 + \frac{3\psi_1 k^2}{n^2}\right)^{t_2}.$$

Proof of Claim 4.8.13. Let q, q' be the scalings before and after one execution of $s_{uv} \leftarrow s_{uv}\cdot 2$. Let ℓ be the number of nonzero eigenvalues of N^q between $(0, \zeta]$, and let $\ell_2 = \min(\ell, \ell_0)$. Then we have $\Upsilon(q) = \det_{\ell_2}(N^q)\zeta^{\ell_0-\ell_2}$, and, by Corollary 4.8.7, $\Upsilon(q') \leqslant \det_{\ell_2}(N^{q'})\zeta^{\ell_0-\ell_2}$. Thus it suffices to show

$$\frac{\det_{\ell_2}(N^{q'})}{\det_{\ell_2}(N^q)} \leqslant 1 + \frac{3\psi_1 k^2}{n^2}. \tag{4.41}$$

As in our proofs of Claims 4.8.11, 4.8.12, we also consider the variational characterization in Lemma 4.8.4. Let $X = (D^q)^{\dagger/2}F$ where $F \in \mathbb{R}^{nk\times\ell_2}$'s columns are nonzero bottom eigenvec-

tors of N^q. The denominator of the characterization can only increase. For the numerator we have by matrix determinant lemma

$$\begin{aligned}\frac{\det\left(X^T L^{q'} X\right)}{\det\left(X^T L^q X\right)} =& 1 + 3(b_{uv}^q)^T X \left(X^T L^q X\right)^{-1} X^T (b_{uv}^q) \\ =& 1 + 3(b_{uv}^q)^T \left(D^q\right)^{\dagger/2} F \left(F^T \left(D^q\right)^{\dagger/2} L^q \left(D^q\right)^{\dagger/2} F\right)^{-1} F^T \left(D^q\right)^{\dagger/2} b_{uv}^q \\ \leqslant& 1 + 3H_\zeta^q(u,v) \leqslant 1 + \frac{3\psi_1 k^2}{n^2}. \end{aligned} \tag{4.42}$$

This implies (4.41) and finishes the proof the claim. □

For the first t iterations of the outermost while loop, let t_1 be the total number of iterations executed by the first inner while loop, and t_2 be the total number of iterations executed by the second inner while loop. Then we have $t_1 \leqslant 2t$ by Lemma 4.8.9, and $t_2 \leqslant t + t_1 \leqslant 3t$. Therefore, by Claims 4.8.11, 4.8.12, 4.8.13, we have

$$\frac{\Upsilon(s^1)}{\Upsilon(s^0)} \leqslant \left(1 - \frac{\psi_2 k^2}{4n^2}\right)^t \left(1 - \frac{400\gamma_2^2 k^2}{(1-\zeta)^2 n^2}\right)^{-2t} \left(1 + \frac{3\psi_1 k^2}{n^2}\right)^{3t} \leqslant \left(1 - \frac{\psi_2 k^2}{16n^2}\right)^t,$$

where the last inequality follows from $\psi_2 \geqslant 1024\psi_1$ and $\psi_2 \geqslant \frac{1024\gamma_2^2}{(1-\zeta)^2}$. □

Notice that by design, after `whack-a-mole-II` terminates, each edge (u,v) with $s_{uv} < s_{uv}^0$ satisfies either $R^s(u,v) \geqslant \frac{\gamma_1 k}{n}$ or $H_\zeta^s(u,v) \geqslant \frac{\psi_1 k^2}{n^2}$. We show that the total number of such edges is small.

Lemma 4.8.14. *After `whack-a-mole-II` terminates, the number of edges (u,v) with $s_{uv} < s_{uv}^0$ such that $R^s(u,v) \geqslant \frac{\gamma_1 k}{n}$ is at most $\frac{n \cdot \mathrm{rank}(D^s)}{\gamma_1 k}$.*

Proof. Since $R^s(u,v)$'s are leverage scores, we have

$$\sum_{(u,v)\in E} R^s(u,v) = \mathrm{rank}\left(D_s\right).$$

Then the desired bound follows. □

Lemma 4.8.15. *After `whack-a-mole-II` terminates, the number of edges (u,v) with $s_{uv} < s_{uv}^0$*

such that $H^s_\zeta(u,v) \geqslant \frac{\psi_1 k^2}{n^2}$ *is at most* $\frac{n^2\gamma_2}{\psi_1(1-\zeta)^2}$.

Proof. Let ℓ be the number of eigenvalues of N^s between $(0,\zeta]$. Let $\lambda_1,\ldots,\lambda_\ell$ be all eigenvalues between $(0,\zeta]$, and let $f_1,\ldots,f_\ell$ be a set of orthonormal eigenvectors. By Theorem 4.6.2, $\ell \leqslant \frac{\gamma_2 k^2}{(1-\zeta)^2}$. Also we have by Proposition 4.2.18 that

$$\sum_{(u,v)\in E} H^s_\zeta(u,v) = \ell.$$

Thus our desired bound follows. □

4.8.2. Analysis of Algorithm 4.8.2

Our analysis of Algorithm 4.8.2 will mostly focus on bounding the total number of deleted edges at Line 13. Since we only delete edges with $(s_{uv}/s'_{uv})^2 \leqslant \rho < 1$, by the termination condition of the second inner while loop of `whack-a-mole-II` we know that each deleted edge satisfies either $R^s(u,v) \geqslant \frac{\gamma_1 k}{n}$ or $H^s_\zeta(u,v) \geqslant \frac{\psi_1 k^2}{n^2}$, where $\gamma_1 = \gamma/16$ and $\psi_1 = \psi/1024$. Thus we will show that the numbers of both types of edges are small.

Consider fixing a while loop iteration of `expander-decomp` where we go the "else" branch. Let s' be the scaling we obtain after we invoke `almost-regular-decomp` at Line 6, s be the scaling returned by `whack-a-mole-II`, and $\hat{s}$ be the scaling obtained after we delete the small weight edges at Line 13 (but before we reset the edge weights at Line 14). Then by an (almost) identical proof to that of Lemma 4.7.5, we have:

Lemma 4.8.16. *The number of edges* $(u,v) \in E$ *with* $(s_{uv}/s'_{uv})^2 \leqslant \rho$ *such that* $R^s(u,v) \geqslant \frac{\gamma_1 \cdot k}{n}$ *is at most*

$$\frac{2n}{\gamma_1 k}\left(\operatorname{rank}\left(D^{s'}\right) - \operatorname{rank}\left(D^{\hat{s}}\right)\right).$$

This means that we can charge the number of deleted edges at Line 13 that satisfy $R^s(u,v) \geqslant \frac{\gamma_1 \cdot k}{n}$ to the rank change of D. We will then bound the number of deleted edges that satisfy $H^s_\zeta(u,v) \geqslant \frac{\psi_1 k^2}{n^2}$ by considering the number of nontrivial (Definition 4.5.20) zero eigenvalues of the normalized

Laplacian, which, by Theorem 4.6.2, is at most $2\gamma k^2$ when the graph is γ-almost regular.

To that end, for a normalized Laplacian matrix N, let $\eta(N)$ denote the number of nontrivial zero eigenvalues of N. Additionally, let s'' denote the scaling we obtain after we invoke the subroutine `almost-regular-decomp` at Line 6 in the *next* iteration of the while loop. The following lemma characterizes how $\eta(N)$ changes after a while loop iteration.

Lemma 4.8.17. *We have*

$$\eta(N^{s''}) \geqslant \eta(N^{s'}) + 1 - \left\lfloor \frac{2\gamma k}{n} \left(\operatorname{rank}\left(D^{s'}\right) - \operatorname{rank}\left(D^{s''}\right) \right) + \sqrt{\frac{1}{16nk}} \right\rfloor . \tag{4.43}$$

Before proving the lemma, we first give the proof of Theorem 4.8.2.

Proof of Theorem 4.8.2. Since we always maintain γ-almost regularity, the number of nontrivial zero eigenvalues can be at most $2\gamma k^2$. Therefore by Lemma 4.8.17 the total number of iterations of the while loop is at most $6\gamma k^2$. Thus, the algorithm terminates in finite time. Also, by the termination condition, the resulting graph G^s must be a (γ, ζ, ψ)-almost regular expander.

By Lemma 4.8.15, the total number of deleted edges with $H^s_\zeta(u, v) \geqslant \frac{\psi_1 k^2}{n^2}$ is at most

$$\frac{n^2\gamma}{\psi_1(1-\zeta)^2} \cdot 6\gamma k^2 \leqslant \frac{10000n^2\gamma^2k^2}{\psi(1-\zeta)^2}.$$

By Lemma 4.8.16 and Theorem 4.7.1, the total number of deleted edges with $R^s(u, v) \geqslant \frac{\gamma_1 \cdot k}{n}$, plus those deleted by `almost-regular-decomp`, is at most

$$\frac{8n}{(\gamma/32)k} \cdot nk = \frac{256n^2}{\gamma}.$$

These two bounds coupled with Lemmas 4.8.14, 4.8.15 imply the desired bound on the number of rescaled edges, and thus finish the proof. □

It then remains to proof Lemma 4.8.17, for which we need the following lemma.

Lemma 4.8.18. *We have*

$$\lambda_{\max}\left(\left(D^{\dagger/2}L^{\hat{s}}D^{\dagger/2}\right)^{\dagger}\right) \leqslant \frac{1}{128nk^3\gamma\cdot\rho}.$$

Proof. Let $\hat{s}_{\min} = \min\{\hat{s}_{uv} : \hat{s}_{uv} \neq 0\}$. Since $\hat{s}$ is obtained from s by zeroing out the edges (u,v) with $(s_{uv}/s'_{uv})^2 \leqslant \rho$, and no $(s_{uv}/s'_{uv})^2$ is in the range $(\rho, \rho\cdot\alpha)$, we have

$$\hat{s}^2_{\min} \geqslant \rho\cdot\alpha\cdot(s'_{\min})^2 \geqslant 128nk^3\gamma\alpha_2\rho,$$

where $s'_{\min} = \min_{s'_{uv}>0} s'_{uv}$ is defined at Line 7, and the last inequality follows from $\alpha \geqslant 128nk^3\gamma\cdot\alpha_2/(s'_{\min})^2$. Let $F = \{(u,v)\in E : \hat{s}_{uv} > 0\}$. Then we have

$$D^{\dagger/2}L^{\hat{s}}D^{\dagger/2} \succeq \hat{s}^2_{\min}D^{\dagger/2}L^{F}D^{\dagger/2} \succeq 128nk^3\gamma\alpha_2\rho D^{\dagger/2}L^{F}D^{\dagger/2},$$

and therefore

$$\left(D^{\dagger/2}L^{\hat{s}}D^{\dagger/2}\right)^{\dagger} \preceq \frac{1}{128nk^3\gamma\alpha_2\rho}\left(D^{\dagger/2}L^{F}D^{\dagger/2}\right)^{\dagger}. \tag{4.44}$$

By definition,

$$\alpha_2 = \max_{F'\subseteq E}\lambda_{\max}\left(\left(D^{\dagger/2}L^{F'}D^{\dagger/2}\right)^{\dagger}\right) \geqslant \lambda_{\max}\left(\left(D^{\dagger/2}L^{F}D^{\dagger/2}\right)^{\dagger}\right).$$

This coupled with (4.44) implies that

$$\left(D^{\dagger/2}L^{\hat{s}}D^{\dagger/2}\right)^{\dagger} \preceq \frac{1}{128nk^3\gamma\rho}I$$

as desired. □

Proof of Lemma 4.8.17. Since s and s' have the same support, we have $\eta(N^{s'}) = \eta(N^s)$. Let $z = \eta(N^s)$. Let $0 = \lambda_1 = \ldots = \lambda_z < \lambda_{z+1} \leqslant \rho$ be the smallest $z+1$ nontrivial eigenvalues of N^s, and let $f_1, \ldots, f_{z+1}$ be a corresponding set of orthonormal, nontrivial eigenvectors. Since $G^{\hat{s}}$ is a

subgraph of G^s, $f_1, \ldots, f_z$ are also zero eigenvectors of $(D^s)^{\dagger/2} L^{\hat{s}} (D^s)^{\dagger/2}$. We first show that f_{z+1} is not in the range of $(D^s)^{\dagger/2} L^{\hat{s}} (D^s)^{\dagger/2}$. In particular, we will show that the projection of f_{z+1} onto the range of $(D^s)^{\dagger/2} L^{\hat{s}} (D^s)^{\dagger/2}$, i.e.

$$\left((D^s)^{\dagger/2} L^{\hat{s}} (D^s)^{\dagger/2}\right)^{\dagger/2} \left((D^s)^{\dagger/2} L^{\hat{s}} (D^s)^{\dagger/2}\right)^{1/2} f_{z+1},$$

has small norm. First note that

$$\begin{aligned}
\left\| \left((D^s)^{\dagger/2} L^{\hat{s}} (D^s)^{\dagger/2}\right)^{1/2} f_{z+1} \right\|^2 =& f_{z+1}^T (D^s)^{\dagger/2} L^{\hat{s}} (D^s)^{\dagger/2} f_{z+1} \\
\leqslant& f_{z+1}^T (D^s)^{\dagger/2} L^{s} (D^s)^{\dagger/2} f_{z+1} \\
=& \lambda_{z+1} \leqslant \rho. \qquad (4.45)
\end{aligned}$$

Then

$$\begin{aligned}
& \left\| \left((D^s)^{\dagger/2} L^{\hat{s}} (D^s)^{\dagger/2}\right)^{\dagger/2} \left((D^s)^{\dagger/2} L^{\hat{s}} (D^s)^{\dagger/2}\right)^{1/2} f_{z+1} \right\|^2 \\
=& f_{z+1}^T \left((D^s)^{\dagger/2} L^{\hat{s}} (D^s)^{\dagger/2}\right)^{1/2} \left((D^s)^{\dagger/2} L^{\hat{s}} (D^s)^{\dagger/2}\right)^{\dagger} \left((D^s)^{\dagger/2} L^{\hat{s}} (D^s)^{\dagger/2}\right)^{1/2} f_{z+1} \\
\leqslant& \lambda_{\max}\left(\left((D^s)^{\dagger/2} L^{\hat{s}} (D^s)^{\dagger/2}\right)^{\dagger}\right) \left\| \left((D^s)^{\dagger/2} L^{\hat{s}} (D^s)^{\dagger/2}\right)^{1/2} f_{z+1} \right\|^2 \\
\leqslant& \lambda_{\max}\left(\left(D^{\dagger/2} L^{\hat{s}} D^{\dagger/2}\right)^{\dagger}\right) \left\| \left((D^s)^{\dagger/2} L^{\hat{s}} (D^s)^{\dagger/2}\right)^{1/2} f_{z+1} \right\|^2 \\
\leqslant& \frac{1}{128nk^3\gamma\rho} \cdot \rho = \frac{1}{128nk^3\gamma}, \qquad (4.46)
\end{aligned}$$

where the last inequality follows from Lemma 4.8.18 and (4.45).

Define a matrix $\mathcal{F} \in \mathbb{R}^{(z+1)\times nk}$ by

$$\mathcal{F} := \begin{pmatrix} f_1^T \\ \vdots \\ f_{z+1}^T \end{pmatrix} \in \mathbb{R}^{(z+1)\times nk}$$

and a function $F : V \to \mathbb{R}^{(z+1)\times k}$ by

$$F(u) = \begin{pmatrix} (f_1)_u^T \\ \vdots \\ (f_{z+1})_u^T \end{pmatrix} \in \mathbb{R}^{(z+1)\times nk}$$

Since G^s is γ-almost regular, we have by Lemma 4.6.4 that $\lambda_{\max}(F(u)F(u)^T) \leqslant \frac{2\gamma k}{n}$ and by Theorem 4.6.2 that $z + 1 \leqslant 2\gamma k^2$.

Now consider projecting the rows of $\mathcal{F}$ twice: (i) project each row onto the null space of the matrix $(D^s)^{\dagger/2}L^{\hat{s}}(D^s)^{\dagger/2}$, and then (ii) project each row onto the range of $(D^s)^{\dagger/2}D^{s''}(D^s)^{\dagger/2}$. Specifically, let us define $\Pi_1, \Pi_2 \in \mathbb{R}^{nk\times nk}$ be the projection matrix onto the range of $(D^s)^{\dagger/2}L^{\hat{s}}(D^s)^{\dagger/2}$ and $(D^s)^{\dagger/2}D^{s''}(D^s)^{\dagger/2}$ respectively, and let $\Pi_1^{\perp}, \Pi_2^{\perp}$ be the projection matrix onto the maximal subspace orthogonal to the range of Π_1 and Π_2 respectively. Then we consider the matrix $\mathcal{F}' := \mathcal{F}\Pi_1^{\perp}\Pi_2$.

In order to prove the lemma, it suffices to show that the rank of $\mathcal{F}'$ is at least

$$z + 1 - \left\lfloor \frac{2\gamma k}{n}\left(\operatorname{rank}\left(D^{s'}\right) - \operatorname{rank}\left(D^{s''}\right)\right) + \sqrt{\frac{1}{16nk}} \right\rfloor, \tag{4.47}$$

because this will imply that the number of zero eigenvalues of $(D^s)^{\dagger/2}L^{s''}(D^s)^{\dagger/2}$ (and hence of $N^{s''}$) is at least $nk - \operatorname{rank}\left(D^{s''}\right) + (4.47)$.

Since

$$\mathcal{F}'(\mathcal{F}')^T = \mathcal{F}\Pi_1^{\perp}\Pi_2\Pi_1^{\perp}\mathcal{F}^T \preceq \mathcal{F}\Pi_1^{\perp}\Pi_1^{\perp}\mathcal{F}^T \preceq \mathcal{F}\mathcal{F}^T = I_{(z+1)\times(z+1)},$$

we know that all singular values of $\mathcal{F}'$ is at most 1. Thus, it suffices to show that

$$\operatorname{Tr}\left(\mathcal{F}'(\mathcal{F}')^T\right) \geqslant z + 1 - \left(\frac{2\gamma k}{n}\left(\operatorname{rank}\left(D^{s'}\right) - \operatorname{rank}\left(D^{s''}\right)\right) + \sqrt{\frac{1}{16nk}}\right),$$

which will imply that the number of nonzero singular values of $\mathcal{F}'$ is at least (4.47). To this end,

let us first write $\mathrm{Tr}\left(\mathcal{F}\Pi_2\mathcal{F}^T\right)$ as

$$\begin{aligned}
&\mathrm{Tr}\left(\mathcal{F}\Pi_2\mathcal{F}^T\right)\\
=&\mathrm{Tr}\left(\mathcal{F}(\Pi_1^\perp+\Pi_1)\Pi_2(\Pi_1^\perp+\Pi_1)\mathcal{F}^T\right)\\
=&\mathrm{Tr}\left(\mathcal{F}\Pi_1^\perp\Pi_2\Pi_1^\perp\mathcal{F}^T\right)+\mathrm{Tr}\left(\mathcal{F}\Pi_1\Pi_2\mathcal{F}^T\right)+\mathrm{Tr}\left(\mathcal{F}\Pi_1^\perp\Pi_2\Pi_1\mathcal{F}^T\right)\\
\leqslant&\mathrm{Tr}\left(\mathcal{F}\Pi_1^\perp\Pi_2\Pi_1^\perp\mathcal{F}^T\right)\\
+&\sqrt{\mathrm{Tr}\left(\mathcal{F}\Pi_1\mathcal{F}^T\right)\mathrm{Tr}\left(\mathcal{F}\Pi_2\mathcal{F}^T\right)}+\sqrt{\mathrm{Tr}\left(\mathcal{F}\Pi_1^\perp\Pi_2\Pi_1^\perp\mathcal{F}^T\right)\mathrm{Tr}\left(\mathcal{F}\Pi_1\mathcal{F}^T\right)},
\end{aligned}\tag{4.48}$$

where the last equality follows from expanding, and the last inequality follows from Cauchy-Schwarz. This combined with (4.46) gives that

$$\begin{aligned}
\mathrm{Tr}\left(\mathcal{F}'(\mathcal{F}')^T\right)=&\mathrm{Tr}\left(\mathcal{F}\Pi_1^\perp\Pi_2\Pi_1^\perp\mathcal{F}^T\right)\\
\geqslant&\mathrm{Tr}\left(\mathcal{F}\Pi_2\mathcal{F}^T\right)-2\sqrt{\frac{z+1}{128nk^3\gamma}}\\
\geqslant&\mathrm{Tr}\left(\mathcal{F}\Pi_2\mathcal{F}^T\right)-2\sqrt{\frac{2\gamma k^2}{128nk^3\gamma}}\\
=&\mathrm{Tr}\left(\mathcal{F}\Pi_2\mathcal{F}^T\right)-\sqrt{\frac{1}{16nk}}.
\end{aligned}\tag{4.49}$$

Using the fact that $\lambda_{\max}(F(u)F(u)^T)\leqslant\frac{2\gamma k}{n}$, we have

$$\mathrm{Tr}\left(\mathcal{F}\Pi_2^\perp\mathcal{F}^T\right)\leqslant\frac{2\gamma k}{n}\left(\mathrm{rank}\left(D^{s'}\right)-\mathrm{rank}\left(D^{s''}\right)\right),$$

and hence

$$\begin{aligned}
\mathrm{Tr}\left(\mathcal{F}\Pi_2\mathcal{F}^T\right)=&\mathrm{Tr}\left(\mathcal{F}\mathcal{F}^T\right)-\mathrm{Tr}\left(\mathcal{F}\Pi_2^\perp\mathcal{F}^T\right)\\
\geqslant&z+1-\frac{2\gamma k}{n}\left(\mathrm{rank}\left(D^{s'}\right)-\mathrm{rank}\left(D^{s''}\right)\right).
\end{aligned}\tag{4.50}$$

Combining (4.49) and (4.50) finishes the proof of the lemma. □

4.9. Expanders are preserved under vertex sampling

4.9.1. Warm-up: ordinary expanders are preserved under vertex sampling

As a warm-up, we start with ordinary unweighted graphs, and prove that an expander with large minimum degree still has good expansion after uniform vertex sampling. We will use the algebraic definition of expansion, which could be translated to combinatorial expansion using Cheeger's inequality [AM85].

Definition 4.9.1 (Expanders)**.** An unweighted graph $G = (V, E)$ is a ζ-expander for some $0 < \zeta \leqslant 1$ if the second smallest eigenvalue of the normalized Laplacian N is $\lambda_2 \geqslant \zeta$.

Note that any ordinary, unweighted graph $G = (V, E)$ with minimum degree $d_{\min}$ can be seen as a 1×1-matrix weighted, $\frac{n}{d_{\min}}$-almost regular graph. Therefore we can apply Lemma 4.6.4 and obtain similar properties of the spectral embedding induced by the bottom eigenvectors of the normalized Laplacian of G.

Definition 4.9.2 (Spectral embeddings of scalar-weighted graphs)**.** Given orthonormal vectors $f_1, \ldots, f_\ell \in \mathbb{R}^n$, define an $\ell \times n$ matrix $\mathcal{F}$ whose rows are transposes of $f_1, \ldots, f_\ell$:

$$\mathcal{F} = \begin{array}{|c|c|c|c|c|} \hline (f_1)_1 & \cdots & (f_1)_u & \cdots & (f_1)_n \\ \hline (f_2)_1 & \cdots & (f_2)_u & \cdots & (f_2)_n \\ \hline \multicolumn{5}{|c|}{\vdots} \\ \hline (f_\ell)_1 & \cdots & (f_\ell)_u & \cdots & (f_\ell)_n \\ \hline \end{array} \in \mathbb{R}^{\ell \times n}.$$

Then define an embedding $F : V \to \mathbb{R}^\ell$ by letting $F(u)$ equal the u^{th} column of $\mathcal{F}$:

$$F(u) = \begin{array}{|c|} \hline (f_1)_u \\ \hline (f_2)_u \\ \hline \vdots \\ \hline (f_\ell)_u \\ \hline \end{array} \in \mathbb{R}^\ell.$$

We call F the spectral embedding induced by $f_1, \ldots, f_\ell$, and $\mathcal{F}$ the embedding matrix induced by $f_1, \ldots, f_\ell$.

By Lemma 4.6.4, we immediately have:

Corollary 4.9.3 (of Lemma 4.6.4)**.** *Let $G = (V, E)$ be an ordinary, unweighted graph with minimum degree $d_{\min}$. Fix a $\delta \in (0, 1)$ and let $0 \leqslant \lambda_1 \leqslant \ldots \leqslant \lambda_\ell \leqslant 1 - \delta$ be all eigenvalues of $N = D^{-1/2} L D^{-1/2}$ that are $\leqslant 1 - \delta$. Let $f_1, \ldots, f_\ell$ be a corresponding set of orthonormal eigenvectors. Let F be the spectral embedding induced by $f_1, \ldots, f_\ell$. Then we have for all $u \in V$*

$$\|F(u)\|^2 \leqslant \frac{1}{\delta^2 d_{\min}}. \tag{4.51}$$

By Theorem 4.6.2 we also have a bound on the number of small eigenvalues:

Corollary 4.9.4 (of Theorem 4.6.2)**.** *Let $G = (V, E)$ be an ordinary, unweighted graph with minimum degree $d_{\min}$. Then for any $\delta \in (0, 1)$, the number of eigenvalues of $N = D^{-1/2} L D^{-1/2}$ that are at most $1 - \delta$ is at most $\frac{n}{\delta^2 d_{\min}}$.*

Theorem 4.9.5. *Let $G = (V, E)$ be a ζ-expander with minimum degree $d_{\min} \geqslant 4 \cdot 10^6 \cdot \zeta^{-1} n^{\frac{32}{\log \log n}}$ for some $\zeta \leqslant \frac{1}{\log n}$. For an $s \geqslant \frac{4 \cdot 10^6 \cdot \zeta^{-1} n^{\frac{32}{\log \log n}}}{d_{\min}} \cdot n$, let C be a uniformly random vertex subset of size s. Then with probability $1 - n^{-7}$, the induced subgraph $G[C]$ is a $\zeta / n^{\frac{1024}{\log \log n}}$-expander with minimum degree at least $\frac{s}{2n} \cdot d_{\min}$.*

This theorem is consequence of applying the following lemma $O(\frac{\log n}{\log \log n})$ times.

Lemma 4.9.6. *Let $G = (V, E)$ be a ζ-expander with minimum degree $d_{\min} \geqslant 2 \cdot 10^6 \cdot \zeta^{-1} \log^{10} n$ for some $\zeta \leqslant \frac{1}{\log n}$. For an $s \geqslant \frac{n}{\log n}$, let C be a uniformly random vertex subset of size s. Then with probability $1 - n^{-8}$, the induced subgraph $G[C]$ is a $\zeta/16$-expander with minimum degree at least $(1 - \frac{1}{2 \log n}) \cdot \frac{s}{n} \cdot d_{\min}$.*

We will prove Lemma 4.9.6 via a matrix Martingale argument, for which end we will need to set up the notions of Cholesky factorization and Schur complements. The following definitions and facts are from [KS16, DKP+17].

Schur complements. Let $L \in \mathbb{R}^{n\times n}$ be the Laplacian matrix of an n-vertex, connected graph G, and let $L(:,u)$ denote the column of L corresponding to vertex u. For a vertex u_1, we define the *Schur complement* of L with respect to u_1 as

$$S^{(1)} = L - \frac{1}{L_{u_1u_1}} L(:,u_1)L(:,u_1)^T \in \mathbb{R}^{n\times n}. \tag{4.52}$$

It is straightforward to see:

Fact 4.9.7. *The entries on row u_1 of $S^{(1)}$ and the entries on column u_1 of $S^{(1)}$ are all zero.*

The following fact is less straightforward, but well known:

Fact 4.9.8. *$S^{(1)}$ is a Laplacian matrix of a graph supported on $V \setminus \{u_1\}$.*

We call the operation of subtracting $\frac{1}{L_{u_1u_1}} L(:,u_1)L(:,u_1)^T$ from L the *elimination of vertex u_1*. Suppose we perform a sequence of eliminations of vertices $u_1, u_2, \ldots, u_t$ for some $1 \leqslant t < n$, and define for each $i \in [t]$

$$\begin{aligned} \alpha_i &= S^{(i-1)}_{u_iu_i} \in \mathbb{R} \\ c_i &= S^{(i-1)}(:,u_i) \in \mathbb{R}^n \\ S^{(i)} &= S^{(i-1)} - \frac{1}{\alpha_i} c_i c_i^T \in \mathbb{R}^{n\times n} \end{aligned}$$

where $S^{(0)} := L$. We call $S^{(t)}$ the *Schur complement* of L with respect to $u_1, \ldots, u_t$. Then similar to Facts 4.9.7 and 4.9.8, we have:

Fact 4.9.9. *The entries on rows $u_1, \ldots, u_t$ of $S^{(t)}$ and the entries on columns $u_1, \ldots, u_t$ of $S^{(t)}$ are all zero. Moreover, $S^{(t)}$ is a Laplacian matrix of a graph supported on $V \setminus \{u_1, \ldots, u_t\}$.*

It is also known that:

Fact 4.9.10. *Changing the order in which we eliminate $u_1, \ldots, u_t$ does not change the resulting Schur complement $S^{(t)}$.*

Let $C = V \setminus \{u_1, \ldots, u_t\}$. We define

$$\mathrm{SC}(L, C) := S^{(t)}_{CC} \in \mathbb{R}^{|C| \times |C|} \tag{4.53}$$

as the *Schur complement of L onto C*, where $S^{(t)}_{CC}$ is $S^{(t)}$ restricted to rows and columns in C. Note that by Fact 4.9.9, $\mathrm{SC}(L, C)$ contains all nonzero entries of $S^{(t)}$. We also write $SC(G, C) = SC(L, C)$. To connect Schur complements to the Laplacian L, we need to introduce partial Cholesky factorization.

Partial Cholesky factorization. Suppose we eliminate a sequence of vertices $u_1, \ldots, u_t$ as above. Let $\mathcal{L}$ be the $n \times t$ matrix whose i^{th} column is c_i, and let $\mathcal{D}$ be the $t \times t$ diagonal matrix with $\mathcal{D}_{ii} = \alpha_i$. Then by adding the matrices subtracted in the elimination steps back to $S^{(t)}$, we have

$$L = S^{(t)} + \sum_{i=1}^{t} \alpha_i c_i c_i^T = S^{(t)} + \mathcal{L}\mathcal{D}\mathcal{L}^T. \tag{4.54}$$

Let us define $F = \{u_1, \ldots, u_t\}$ and $C = V \setminus F$. Let $S = \mathrm{SC}(L, C)$, where we recall that the latter is $S^{(t)}$ restricted to rows and columns in C. Since S contains all nonzero entries of $S^{(t)}$, by writing $\mathcal{L}$ in block form as $\mathcal{L} = \begin{pmatrix} \mathcal{L}_{FF} \\ \mathcal{L}_{CF} \end{pmatrix}$, we get

$$L = \begin{pmatrix} \mathcal{L}_{FF} \\ \mathcal{L}_{CF} \end{pmatrix} \mathcal{D} \begin{pmatrix} \mathcal{L}_{FF} \\ \mathcal{L}_{CF} \end{pmatrix}^T + \begin{pmatrix} 0_{FF} & 0_{FC} \\ 0_{CF} & S \end{pmatrix} = \begin{pmatrix} \mathcal{L}_{FF} & 0 \\ \mathcal{L}_{CF} & I_{CC} \end{pmatrix} \begin{pmatrix} \mathcal{D} & 0 \\ 0 & S \end{pmatrix} \begin{pmatrix} \mathcal{L}_{FF} & 0 \\ \mathcal{L}_{CF} & I_{CC} \end{pmatrix}^T. \tag{4.55}$$

We call (4.55) a *partial Cholesky factorization* of L.

We now state a fact about matrix factorizations of the form in (4.55). For a set of vectors $x_1, \ldots, x_s$, we will write $\Pi_{x_1,\ldots,x_s}$ to denote the projection matrix onto their linear span, and $\Pi^{\perp}_{x_1,\ldots,x_s}$ to denote the projection matrix onto the maximal subspace orthogonal to their linear span.

Fact 4.9.11. *Given an $n \times n$ positive semidefinite matrix M that can be factorized as*

$$M = \begin{pmatrix} \mathcal{L}_{FF} \\ \mathcal{L}_{CF} \end{pmatrix} \mathcal{D} \begin{pmatrix} \mathcal{L}_{FF} \\ \mathcal{L}_{CF} \end{pmatrix}^T + \begin{pmatrix} 0_{FF} & 0_{FC} \\ 0_{CF} & S \end{pmatrix} = \begin{pmatrix} \mathcal{L}_{FF} & 0 \\ \mathcal{L}_{CF} & I_{CC} \end{pmatrix} \begin{pmatrix} \mathcal{D} & 0 \\ 0 & S \end{pmatrix} \begin{pmatrix} \mathcal{L}_{FF} & 0 \\ \mathcal{L}_{CF} & I_{CC} \end{pmatrix}^T$$

for some bi-partition (C, F) of $[n]$, such that (i) $\mathcal{L}_{FF} \in \mathbb{R}^{|F|\times|F|}$ is a full-rank matrix; (ii) $\mathcal{L}_{CF} \in \mathbb{R}^{|C|\times|F|}$; (iii) $\mathcal{D}$ is a diagonal matrix of non-negative entries. Let $f_1, \ldots, f_z$ be a basis of the null space of M. Then

$$S^\dagger = \Pi^\perp_{(f_1)_C,\ldots,(f_z)_C} (M^\dagger)_{CC} \Pi^\perp_{(f_1)_C,\ldots,(f_z)_C},$$

where $(M^\dagger)_{CC}$ is $M^\dagger$ restricted to rows and columns in C.

As a direct corollary of the above fact, we have:

Fact 4.9.12 (Corollary of Fact 4.9.11)**.** *Let $f_1 \in \mathbb{R}^n$ be the all-one vector. For any $C \subseteq V$,*

$$\mathrm{SC}(L, C) = \Pi^\perp_{(f_1)_C} (L^\dagger)_{CC} \Pi^\perp_{(f_1)_C}.$$

Below we will need to use the following lemma, which is a direct consequence of the Matrix Chernoff bound (Theorem 2.3.4).

Lemma 4.9.13. *Let $f_1, \ldots, f_\ell \in \mathbb{R}^n$ be any ℓ orthonormal vectors. Define $F : [n] \to \mathbb{R}^\ell$ by letting $F(i) = ((f_1)_1, \ldots, (f_\ell)_i)^T \in \mathbb{R}^\ell$. Suppose for any $i \in [n]$,*

$$\|F(i)\|^2 \leqslant \rho$$

for some ρ. For an $s \geqslant 100\rho n \log n$, let $C \subseteq [n]$ be a uniformly random subset of indices of size s. Then we have with probability $1 - n^{-10}$ that

$$\frac{1}{2} I_{\ell\times\ell} \preceq \frac{n}{s} \left(\sum_{i\in C} F(i) F(i)^T \right) \preceq 2 I_{\ell\times\ell}$$

and therefore

$$\sum_{j=1}^{\ell}(f_j)_C(f_j)_C^T \preceq \frac{2s}{n}I_{n\times n}.$$

The following lemma shows that the Schur complement onto a random vertex subset is a better expander than the original graph. Here we use $\Pi_x^{\perp}$ to denote the projection matrix onto the maximal subspace orthogonal to vector x.

Lemma 4.9.14. *Let $G=(V,E)$ be a ζ-expander with minimum degree $d_{\min}\geqslant 2\cdot 10^6\log^5 n$ where $\zeta\leqslant\frac{1}{\log n}$. Let the eigenvalues of N_G be $0=\lambda_1<\zeta\leqslant\lambda_2\leqslant\ldots\leqslant\lambda_n$ and let $f_1,\ldots,f_n$ be a set of corresponding orthonormal eigenvectors. For an $s\geqslant\frac{n}{\log n}$, let C be a uniformly random vertex subset of size s. Then with probability $1-n^{-10}$*

$$D_{CC}^{-1/2}\mathrm{SC}(L_G,C)D_{CC}^{-1/2}\succeq\frac{n}{4s}\cdot\zeta\cdot\Pi_{(f_1)_C}^{\perp}.$$

Proof. By eliminating vertices outside of C, we get a partial Cholesky factorization of L_G:

$$L_G=\begin{pmatrix}\mathcal{L}_{FF} & 0\\ \mathcal{L}_{CF} & I_{CC}\end{pmatrix}\begin{pmatrix}\mathcal{D} & 0\\ 0 & \mathrm{SC}(L_G,C)\end{pmatrix}\begin{pmatrix}\mathcal{L}_{FF} & 0\\ \mathcal{L}_{CF} & I_{CC}\end{pmatrix}^T.$$

By multiplying $D^{-1/2}$ on both sides, we then get a factorization of N_G:

$$\begin{aligned}N_G&=\begin{pmatrix}D_{FF}^{-1/2}\mathcal{L}_{FF} & 0\\ D_{CC}^{-1/2}\mathcal{L}_{CF} & D_{CC}^{-1/2}\end{pmatrix}\begin{pmatrix}\mathcal{D} & 0\\ 0 & \mathrm{SC}(L_G,C)\end{pmatrix}\begin{pmatrix}D_{FF}^{-1/2}\mathcal{L}_{FF} & 0\\ D_{CC}^{-1/2}\mathcal{L}_{CF} & D_{CC}^{-1/2}\end{pmatrix}^T\\ &=\begin{pmatrix}D_{FF}^{-1/2}\mathcal{L}_{FF} & 0\\ D_{CC}^{-1/2}\mathcal{L}_{CF} & I_{CC}\end{pmatrix}\begin{pmatrix}\mathcal{D} & 0\\ 0 & D_{CC}^{-1/2}\mathrm{SC}(L_G,C)D_{CC}^{-1/2}\end{pmatrix}\begin{pmatrix}D_{FF}^{-1/2}\mathcal{L}_{FF} & 0\\ D_{CC}^{-1/2}\mathcal{L}_{CF} & I_{CC}\end{pmatrix}^T.\end{aligned}$$

Then by Fact 4.9.11,

$$\left(D_{CC}^{-1/2}\mathrm{SC}(L_G,C)D_{CC}^{-1/2}\right)^{\dagger}=\Pi_{(f_1)_C}^{\perp}\left(\sum_{i=2}^{n}\frac{1}{\lambda_i}(f_i)_C(f_i)_C^T\right)\Pi_{(f_1)_C}^{\perp}.$$

Let ℓ be such that $\lambda_\ell \leqslant \max\left\{\frac{n}{2s}, 1\right\} \cdot \zeta < \lambda_{\ell+1}$. Define the embedding $F : V \to \mathbb{R}^\ell$ as

$$F(u) = \begin{pmatrix} (f_1)_u \\ (f_2)_u \\ \vdots \\ (f_\ell)_u \end{pmatrix} \in \mathbb{R}^\ell.$$

By $s \geqslant n/\log n$ and $\zeta \leqslant 1/\log n$, we have $\frac{n}{2s} \cdot \zeta \leqslant 1/2$, and thus by Corollary 4.9.3 we have that for each $u \in V$

$$\|F(u)\|^2 \leqslant \frac{4}{d_{\min}} \leqslant 10^{-4} \cdot \log^{-3} n.$$

Then we have

$$\begin{aligned}
\left(D_{CC}^{-1/2}\mathrm{SC}(L_G, C)D_{CC}^{-1/2}\right)^\dagger =& \Pi^\perp_{(f_1)_C}\left(\sum_{i=2}^{n} \frac{1}{\lambda_i}(f_i)_C(f_i)_C^T\right)\Pi^\perp_{(f_1)_C} \\
\preceq& \Pi^\perp_{(f_1)_C}\left(\sum_{i=2}^{\ell} \frac{1}{\zeta}(f_i)_C(f_i)_C^T + \sum_{i=\ell+1}^{n} \frac{2s}{n} \cdot \frac{1}{\zeta}(f_i)_C(f_i)_C^T\right)\Pi^\perp_{(f_1)_C} \\
\preceq& \Pi^\perp_{(f_1)_C}\left(\sum_{i=2}^{\ell} \frac{1}{\zeta}(f_i)_C(f_i)_C^T + \frac{2s}{n} \cdot \frac{1}{\zeta} I\right)\Pi^\perp_{(f_1)_C} \\
\preceq& \Pi^\perp_{(f_1)_C}\left(\frac{2s}{n} \cdot \frac{1}{\zeta} I + \frac{2s}{n} \cdot \frac{1}{\zeta} I\right)\Pi^\perp_{(f_1)_C} \\
=& \frac{4s}{n} \cdot \frac{1}{\zeta}\Pi^\perp_{(f_1)_C},
\end{aligned}$$

where the third line follows from $\lambda_{\max}(\sum_{i=\ell+1}^{n}(f_i)_C(f_i)_C^T) \leqslant \lambda_{\max}(\sum_{i=1}^{n} f_i f_i^T) = 1$, and the second to last line holds with probability $1 - n^{-10}$ by Lemma 4.9.13. By inverting both sides, we then get

$$D_{CC}^{-1/2}\mathrm{SC}(L_G, C)D_{CC}^{-1/2} \succeq \frac{n}{4s} \cdot \zeta \cdot \Pi^\perp_{(f_1)_C},$$

as desired. □

We now define the notion of graph squaring, which will be useful in setting up our martingale.

Definition 4.9.15 (Graph squaring)**.** For a graph $G = (V, E)$ with degree matrix D and adjacency matrix A, define the square of G by

$$L_{G^2} \stackrel{\text{def}}{=} D - AD^{-1}A.$$

Fact 4.9.16 ([CCL+15, JKPS17])**.** $L_{G^2} \preceq 2L_G$ *for any graph* G*.*

Below, we will write $\mathrm{SC}_n(L, C)$ or $\mathrm{SC}_n(G, C)$ to denote an $n \times n$ matrix obtained by augmenting $\mathrm{SC}(L, C)$ to $n \times n$ by adding zeros on rows and columns in $V \setminus C$. We will also abuse the notation a bit and write $L_{G[C]}$ to denote an $n \times n$ matrix obtained by augmenting $L_{G[C]}$ to $n \times n$ by adding zeros on rows and columns in $V \setminus C$.

Proposition 4.9.17. $\sum_{u \in V} \mathrm{SC}_n(L_G, V \setminus \{u\}) = L_{G^2} + (n-2)L_G$ *for any graph* G*.*

Proof. Notice that by definition

$$\begin{aligned}
\sum_{u \in V} \mathrm{SC}_n(L, V \setminus \{u\}) =& \sum_{u \in V} \left(L_G - \frac{1}{L_{uu}} L_G(:, u) L_G(:, u)^T \right) \\
=& nL_G - L_G D^{-1} L_G \\
=& n(D - A) - (D - A)D^{-1}(D - A) \\
=& nD - nA - (D - 2A + AD^{-1}A) \\
=& (n-2)(D - A) + D - AD^{-1}A = (n-2)L_G + L_{G^2}
\end{aligned}$$

as desired. □

Let $v_1, \ldots, v_n$ be a uniformly random permutation of vertices in V. Define

$$V_i = \{v_1, \ldots, v_i\}.$$

Consider the following sequence of matrices:

$$
\begin{aligned}
X_0 &= L_G \\
X_1 &= \frac{1}{d_{v_1}} L_G(:, v_1) L_G(:, v_1)^T + \left(1 + \frac{2}{n-2}\right) L_{G[V-V_1]} + \frac{1}{n}\left(L_{G^2} - 2L_G\right) \\
X_2 &= \frac{1}{d_{v_1}} L_G(:, v_1) L_G(:, v_1)^T + \frac{1}{n}\left(L_{G^2} - 2L_G\right) + \left(1 + \frac{2}{n-2}\right) \cdot \\
&\quad \left(\frac{1}{d_{v_2}^{G[V-V_1]}} L_{G[V-V_1]}(:, v_2) L_{G[V-V_1]}(:, v_2)^T + \right. \\
&\quad \left. \left(1 + \frac{2}{n-3}\right) L_{G[V-V_2]} + \frac{1}{n-1}\left(L_{G[V-V_1]^2} - 2L_{G[V-V_1]}\right) \right)
\end{aligned}
$$

X_{i+1} is obtained by replacing the $L_{G[V-V_i]}$ in X_i by

$$
\frac{1}{d_{v_{i+1}}^{G[V-V_i]}} L_{G[V-V_i]}(:, v_{i+1}) L_{G[V-V_i]}(:, v_{i+1})^T + \tag{4.56}
$$

$$
\left(1 + \frac{2}{n-2-i}\right) L_{G[V-V_{i+1}]} + \frac{1}{n-i}\left(L_{G[V-V_i]^2} - 2L_{G[V-V_i]}\right). \tag{4.57}
$$

Here, we have used d_v^H to denote the degree of vertex v in graph H.

Lemma 4.9.18. *$X_0, X_1, X_2, \ldots$ is a matrix-valued martingale.*

Proof. It suffices to prove that

$$
\mathbb{E}_{v_{i+1}}\left[(4.56) \mid V_i\right] = L_{G[V-V_i]}.
$$

Let us calculate the LHS term by term.

$$
\begin{aligned}
&\mathbb{E}_{v_{i+1}}\left[\frac{1}{d_{v_{i+1}}^{G[V-V_i]}} L_{G[V-V_i]}(:, v_{i+1}) L_{G[V-V_i]}(:, v_{i+1})^T \mid V_i\right] \\
=&\frac{1}{n-i} L_{G[V-V_i]} D_{G[V-V_i]}^{-1} L_{G[V-V_i]} \\
=&\frac{1}{n-i}\left(D_{G[V-V_i]} - 2A_{G[V-V_i]} + A_{G[V-V_i]} D_{G[V-V_i]}^{-1} A_{G[V-V_i]}\right)
\end{aligned}
$$

$$\mathbb{E}_{v_{i+1}}\left[\left(1+\frac{2}{n-2-i}\right)L_{G[V-V_{i+1}]} \mid V_i\right] = \frac{n-2-i}{n-i}L_{G[V-V_i]} + \frac{2}{n-i}L_{G[V-V_i]} = L_{G[V-V_i]}$$

$$\begin{aligned}
&\mathbb{E}_{v_{i+1}}\left[\frac{1}{n-i}\left(L_{G[V-V_i]^2} - 2L_{G[V-V_i]}\right) \mid V_i\right] \\
=&\frac{1}{n-i}\left(L_{G[V-V_i]^2} - 2L_{G[V-V_i]}\right) \\
=&\frac{1}{n-i}\left(D_{G[V-V_i]} - A_{G[V-V_i]}D^{-1}_{G[V-V_i]}A_{G[V-V_i]}\right) - \frac{2}{n-i}L_{G[V-V_i]}.
\end{aligned}$$

One can verify that these three add up to $L_{G[V-V_i]}$. □

To analyze this matrix-valued martingale, we will resort to the following theorem:

Theorem 4.9.19 (Matrix Freedman Inequality [Tro11])**.** *Consider a zero-mean matrix martingale* $\{Y_i : i = 0, 1, \ldots\}$ *whose values are symmetric matrices with dimensional* n*, and let* $\{Z_i : i = 0, 1, \ldots\}$ *be the difference sequence such that* $Z_i = Y_{i+1} - Y_i$*. Assume the difference sequence is uniformly bounded in the sense that for any* $i = 0, 1, \ldots$

$$\lambda_{\max}(Z_i) \leqslant R,$$

where $\lambda_{\max}(Z_i)$ *is the maximum absolute value of any eigenvalue of* Z_i*. Define the predictable quadratic variation process of the martingale as*

$$W_i \stackrel{\text{def}}{=} \sum_{j=0}^{i} \mathbb{E}\left[Z_j^2 \mid Y_1, \ldots, Y_j\right].$$

Then for all $i \geqslant 0$ *and* $\sigma^2 > 0$*,*

$$\Pr\left[\exists i \geqslant 0 : \lambda_{\max}(Y_i) \geqslant t \quad \text{and} \quad \|W_i\| \leqslant \sigma^2\right] \leqslant d \cdot \exp\left\{-\frac{t^2/2}{\sigma^2 + Rt/3}\right\}.$$

We now prove Lemma 4.9.6.

Lemma 4.9.20. *Let* $G = (V, E)$ *be a* ζ*-expander with minimum degree* $d_{\min} \geqslant 2 \cdot 10^6 \cdot \zeta^{-1} \log^{10} n$ *for some* $\zeta \leqslant \frac{1}{\log n}$*. For an* $s \geqslant \frac{n}{\log n}$*, let* C *be a uniformly random vertex subset of size* s*. Then with*

probability $1 - n^{-8}$*, the induced subgraph* $G[C]$ *is a* $\zeta/16$*-expander with minimum degree at least* $(1 - \frac{1}{2\log n}) \cdot \frac{s}{n} \cdot d_{\min}$.

Proof. The minimum degree guarantee follows from a direct application of Chernoff bounds. We then focus on showing that $G[C]$ is a $\zeta/4$-expander.

Define $Y_i = L_G^{\dagger/2} X_i L_G^{\dagger/2} - L_G^{\dagger/2} L_G L_G^{\dagger/2}$. Since $X_0, X_1, \ldots$ is a martingale, so is $Y_0, Y_1, \ldots$, by linearity of expectation. Moreover, since $X_0 = L_G$, $Y_0, Y_1, \ldots$ has zero mean. Consider the first $n - s + 1$ terms $Y_0, Y_1, Y_2, \ldots, Y_{n-s}$. First let us calculate the difference sequence Z_i:

$$\begin{aligned} Z_i =& Y_{i+1} - Y_i \\ =& \prod_{j=1}^{i} \left(1 + \frac{2}{n-1-j}\right) \cdot L_G^{\dagger/2} \\ & \left(- \mathrm{SC}_n\left(G[V - V_i], V - V_{i+1}\right) + \left(1 + \frac{2}{n-2-i}\right) L_{G[V-V_{i+1}]} + \right. \\ & \left. \frac{1}{n-i}\left(L_{G[V-V_i]^2} - 2L_{G[V-V_i]}\right)\right) L_G^{\dagger/2}, \end{aligned}$$

where we have used Definitions (4.52),(4.53), and recall that $\mathrm{SC}_n(G, C)$ is obtained by augmenting $\mathrm{SC}(G, C)$ to $n \times n$ by adding zeros. We then consider to bound the maximum (in absolute value) eigenvalue of Z_i. For the last term, we have

$$\lambda_{\max}\left(L_G^{\dagger/2} \frac{1}{n-i}\left(L_{G[V-V_i]^2} - 2L_{G[V-V_i]}\right) L_G^{\dagger/2}\right) \leqslant \frac{4}{n-i} \leqslant \frac{4}{s} \leqslant \frac{4\log n}{d_{\min}} \leqslant \frac{4}{10^6 \log^4 n}, \tag{4.58}$$

where the first inequality follows from $L_{G[V-V_i]^2} \preceq 2L_{G[V-V_i]} \preceq 2L_G$ by Fact 4.9.16. Now to bound the first two terms, we first rearrange them as

$$\left(-\mathrm{SC}_n\left(G[V - V_i], V - V_{i+1}\right) + L_{G[V-V_{i+1}]}\right) + \frac{2}{n-2-i} L_{G[V-V_{i+1}]}. \tag{4.59}$$

Then for the second term of (4.59), we have $L_{G[V-V_{i+1}]} \preceq L_G$, and hence

$$\lambda_{\max}\left(L_G^{\dagger/2} \frac{2}{n-2-i} L_{G[V-V_{i+1}]} L_G^{\dagger/2}\right) \leqslant \frac{2}{n-2-i} \leqslant \frac{2}{s-2} \leqslant \frac{4}{s} \leqslant \frac{4\log n}{d_{\min}} \leqslant \frac{4}{10^6 \log^4 n}. \tag{4.60}$$

We then claim that the first term of (4.59) can be bounded by

$$\lambda_{\max}\left(L_G^{\dagger/2}\left(\mathrm{SC}_n\left(G[V-V_i],V-V_{i+1}\right)-L_{G[V-V_{i+1}]}\right)L_G^{\dagger/2}\right)\leqslant\frac{1}{d_{\min}\zeta}\leqslant\frac{1}{8\cdot 10^6\log^5 n}. \tag{4.61}$$

To see why this is the case, let us define an edge set $E_{i+1}=\{(v_{i+1},w)\in G[V-V_i]\}$ and write $L_{E_{i+1}}$ to denote the Laplacian of the subgraph of G induced by E_{i+1}. Notice that

$$L_{E_{i+1}}(:,v_{i+1})=L_{G[V-V_i]}(:,v_{i+1}).$$

Therefore, we have

$$\mathrm{SC}_n\left(G[V-V_i],V-V_{i+1}\right)-L_{G[V-V_{i+1}]}=\mathrm{SC}_n\left(L_{E_{i+1}},V-V_{i+1}\right).$$

Therefore to prove (4.61), it suffices to show that for any x such that $x\in\mathrm{range}(\mathrm{SC}_n(L_{E_{i+1}},V-V_{i+1}))$, we have

$$x^TL^{\dagger}x\leqslant\frac{1}{\zeta d_{\min}}\cdot x^T\mathrm{SC}_n\left(L_{E_{i+1}},V-V_{i+1}\right)^{\dagger}x=\frac{1}{\zeta d_{\min}}\cdot x^TL_{E_{i+1}}^{\dagger}x, \tag{4.62}$$

where the equality follows from Fact 4.9.12. Since G is a ζ-expander with minimum degree $d_{\min}$, we know that $x^TL^{\dagger}x\leqslant\frac{1}{\zeta d_{\min}}\|x\|^2$. On the other hand, as $G[E_{k+1}]$ is a star graph, and $x\in\mathrm{range}(\mathrm{SC}_n(L_{E_{i+1}},V-V_{i+1}))$ implies that x is supported on $V-V_{i+1}$, we have $x^TL_{E_{i+1}}^{\dagger}x=\|x\|^2$. This proves our desired inequality (4.62).

We also note that

$$\prod_{j=1}^{i}\left(1+\frac{2}{n-1-j}\right)=\frac{n}{n-2}\frac{n-1}{n-3}\cdots\frac{n+1-i}{n-1-i}=\frac{n(n-1)}{(n-i)(n-1-i)}\leqslant\left(\frac{n}{s}\right)^2\leqslant\log^2 n.$$

Therefore we have

$$\lambda_{\max}(Z_i)\leqslant\frac{1}{1000\cdot\log^2 n}. \tag{4.63}$$

Fixing a j, we then consider bounding the spectral norm of $\mathbb{E}\left[Z_j^2 \mid Y_1, \ldots, Y_{j-1}\right]$.

$$
\begin{aligned}
&\mathbb{E}\left[Z_j^2 \mid Y_1, \ldots, Y_{j-1}\right] \\
\preceq &\frac{2}{n-j} \cdot \left(\frac{n}{n-1-j}\right)^4 \\
&\sum_{v \in V - V_j} \left(L_G^{\dagger/2}\left(\mathrm{SC}_n(G[V-V_j], V-V_j-\{v\}) - L_{G[V-V_j-\{v\}]}\right) L_G^{\dagger/2})^2 + \right. \\
&\frac{2}{n-j} \cdot \left(\frac{n}{n-1-j}\right)^4 \\
&\sum_{v \in V - V_j} \left(L_G^{\dagger/2}\left(\frac{2}{n-2-j} L_{G[V-V_j-\{v\}]} + \frac{1}{n-j}\left(L_{G[V-V_j]^2} + 2L_{G[V-V_j]}\right)\right) L_G^{\dagger/2}\right)^2 \\
\preceq &\frac{2\log^5 n}{n-j} \frac{1}{\zeta d_{\min}} \sum_{v \in V - V_j} L_G^{\dagger/2}\left(\mathrm{SC}_n(G[V-V_j], V-V_j-\{v\}) - L_{G[V-V_j-\{v\}]}\right) L_G^{\dagger/2} \\
+ &\frac{2\log^5 n}{n-j} \cdot |V - V_j| \cdot \left(\frac{8}{s}\right)^2 I_{n\times n} \qquad \text{(by (4.61),(4.60),(4.58))} \\
\preceq &\frac{2\log^6 n}{n\zeta d_{\min}} L_G^{\dagger/2} L_{G[V-V_j]^2} L_G^{\dagger/2} + \frac{10000\log^7 n}{n^2} I_{n\times n} \\
&\qquad \text{(by Proposition 4.9.17 and } n - j \geqslant s \geqslant n/\log n) \\
\preceq &\frac{64}{2\cdot 10^6 \cdot n\log^3 n} I_{n\times n} + \frac{10000\log^7 n}{n^2} I_{n\times n} \qquad \text{(Fact 4.9.16 and } d_{\min} \text{ lower bound)} \\
\preceq &\frac{1}{1000 n \log^2 n} I_{n\times n}.
\end{aligned}
$$

Therefore we have

$$
\lambda_{\max}\left(\mathbb{E}\left[Z_j^2 \mid Y_1, \ldots, Y_{j-1}\right]\right) \leqslant \frac{1}{1000 n \log^2 n}
$$

and

$$
\begin{aligned}
\lambda_{\max}\left(\sum_{j=0}^{n-s-1} \mathbb{E}\left[Z_j^2 \mid Y_1, \ldots, Y_{j-1}\right]\right) &\leqslant \sum_{j=0}^{n-s-1} \lambda_{\max}\left(\mathbb{E}\left[Z_j^2 \mid Y_1, \ldots, Y_{j-1}\right]\right) \\
&\leqslant n \cdot \frac{1}{1000 n \log^2 n} \\
&= \frac{1}{1000 \log^2 n}. \qquad (4.64)
\end{aligned}
$$

We now invoke Theorem 4.9.19 with $R = \frac{1}{1000\log^2 n}$, $\sigma^2 = \frac{1}{1000\log^2 n}$, and $t = 1/2$, and get

$$\Pr\left[\exists i \in [0, n-s] : \lambda_{\max}(Y_i) \geqslant \frac{1}{10\log n}\right] \leqslant n \cdot \exp\left(-\frac{\frac{1}{8}}{\frac{1}{1000\log^2 n} + \frac{1}{6000\log^2 n}}\right) \leqslant n^{-10}.$$

This coupled with Fact 4.9.11 implies that

$$L_{G[C]} \succeq \frac{1}{2} \cdot \frac{s^2}{n^2} \cdot \mathrm{SC}(L_G, C).$$

Then we have

$$\begin{aligned}\lambda_2\left(D_{CC}^{-1/2} L_{G[C]} D_{CC}^{-1/2}\right) \geqslant& \frac{1}{2} \cdot \frac{s^2}{n^2} \cdot \lambda_2\left(D_{CC}^{-1/2} \mathrm{SC}(L_G, C) D_{CC}^{-1/2}\right) \\ \geqslant& \frac{1}{8} \cdot \frac{s}{n} \cdot \zeta,\end{aligned}$$

where the last inequality holds with probability $1 - n^{-10}$ by Lemma 4.9.14. Note that, by a Chernoff bound, we have with probability $1 - n^{-10}$ that

$$\frac{1}{2} D_{CC} \preceq \frac{n}{s} D_{G[C]} \preceq 2 D_{CC}.$$

These two combined imply that $G[C]$ is a $\zeta/16$-expander with probability $1 - 3 \cdot n^{-10}$. □

4.9.2. Matrix-weighted expanders are preserved under vertex sampling

We now introduce Schur complements and Cholesky factorization for Laplacian matrices of $k \times k$ matrix-weighted graphs.

Schur complements. Let $L \in \mathbb{R}^{nk \times nk}$ be the Laplacian of a $k \times k$ matrix-weighted graph G, and let $L(:, u) \in \mathbb{R}^{nk \times k}$ to denote the column block of L corresponding to vertex u. For a vertex u_1, we define the *Schur complement* of L with respect to u_1 as

$$S^{(1)} = L - L(:, u_1) L_{u_1 u_1}^{\dagger} L(:, u_1)^T. \tag{4.65}$$

It is straightforward to see:

Fact 4.9.21. *The entries on row block u_1 of $S^{(1)}$ and the entries on column block u_1 of $S^{(1)}$ are all zero.*

We call the operation of subtracting $L(:, u_1)L^{\dagger}_{u_1u_1}L(:, u_1)^T$ from L the *elimination of vertex* u_1. Unlike the ordinary graph case, if we eliminate a sequence of vertices one by one, the resulting matrix is not necessarily the Laplacian of another $k \times k$ matrix-weighted graph. So we alternatively define the *Schur complement with respect to a vertex set* F as

$$\mathrm{SC}(L, C) = L_{CC} - L_{CF}L^{\dagger}_{FF}L_{FC}, \tag{4.66}$$

where $C := V \setminus F$. We also call $\mathrm{SC}(L, C)$ the *Schur complement of L (or G) onto C*. Note that when $F = \{u_1\}$ consists of only a single vertex, we have $S^{(1)}_{CC} = \mathrm{SC}(L, C)$. Also note that when $k = 1$, this definition matches the alternative definition of Schur complements in ordinary graphs [KLP$^+$16]. We then connect $\mathrm{SC}(L, C)$ to the Laplacian L by introducing partial Cholesky factorization.

Partial Cholesky factorization. We show that L can be factorized in a similar way to [KLP$^+$16]. The proof of the following proposition is deferred to Appendix B.5.

Proposition 4.9.22. *We have*

$$L = \begin{pmatrix} I_{FF} & 0 \\ L_{CF}L^{\dagger}_{FF} & I_{CC} \end{pmatrix} \begin{pmatrix} L_{FF} & 0 \\ 0 & \mathrm{SC}(L, C) \end{pmatrix} \begin{pmatrix} I_{FF} & 0 \\ L_{CF}L^{\dagger}_{FF} & I_{CC} \end{pmatrix}^T. \tag{4.67}$$

We call (4.67) a *partial Cholesky factorization* of L. We now state a fact about matrix factorizations of the form in (4.67). As in the previous subsection, for a set of vectors $x_1, \ldots, x_s$, we will write $\Pi_{x_1,\ldots,x_s}$ to denote the projection matrix onto their linear span, and $\Pi^{\perp}_{x_1,\ldots,x_s}$ to denote the projection matrix onto the maximal subspace orthogonal to their linear span.

Fact 4.9.23 (See e.g. [KLP$^+$16])**.** *Given an $nk \times nk$ positive semidefinite matrix M divided into*

$k \times k$ blocks that can be factorized as

$$M = \begin{pmatrix} I_{FF} & 0 \\ \mathcal{L}_{CF} & I_{CC} \end{pmatrix} \begin{pmatrix} \mathcal{L}_{FF} & 0 \\ 0 & S \end{pmatrix} \begin{pmatrix} I_{FF} & 0 \\ \mathcal{L}_{CF} & I_{CC} \end{pmatrix}^T$$

for some bi-partition (C, F) of $[n]$. Let $f_1, \ldots, f_z \in \mathbb{R}^{nk}$ be a basis of the null space of M. Then

$$S^\dagger = \Pi^\perp_{(f_1)_C, \ldots, (f_z)_C} (M^\dagger)_{CC} \Pi^\perp_{(f_1)_C, \ldots, (f_z)_C},$$

where $(M^\dagger)_{CC}$ is $M^\dagger$ restricted to row and column blocks in C.

As a direct corollary of the above fact, we have:

Fact 4.9.24 (Corollary of Fact 4.9.11)**.** *Let $f_1, \ldots, f_z \in \mathbb{R}^{nk}$ be a basis of the null space of the matrix L. For any $C \subseteq V$,*

$$\mathrm{SC}(L, C) = \Pi^\perp_{(f_1)_C, \ldots, (f_z)_C} (L^\dagger)_{CC} \Pi^\perp_{(f_1)_C, \ldots, (f_z)_C}.$$

Below we will need to use the following lemma, which is a direct consequence of the Matrix Chernoff bound (Theorem 2.3.4).

Lemma 4.9.25. *Let $f_1, \ldots, f_\ell \in \mathbb{R}^{nk}$ be any ℓ orthonormal vectors. Define $F : [n] \to \mathbb{R}^{\ell \times k}$ by*

$$F(u) = \begin{pmatrix} (f_1)_u^T \\ \vdots \\ (f_\ell)_u^T \end{pmatrix} \in \mathbb{R}^{\ell \times k}.$$

Suppose for any $u \in [n]$

$$\lambda_{\max}\left(F(u)^T F(u)\right) \leqslant R.$$

For an $s \geqslant 100Rn \log n$, let $C \subseteq [n]$ be a uniformly random subset of indices of size s. Then we

have with probability $1 - n^{-10}$ *that*

$$\frac{1}{2} I_{\ell \times \ell} \preceq \frac{n}{s} \left(\sum_{u \in C} F(u) F(u)^T \right) \preceq 2 I_{\ell \times \ell}$$

and therefore

$$\sum_{j=1}^{\ell} (f_j)_C (f_j)_C^T \preceq \frac{2s}{n} I_{nk \times nk}.$$

Definition 4.9.26 (Subgraph preservation)**.** Let $G = (V, E)$ be a $k \times k$ matrix-weighted graph. Let $\lambda_1, \ldots, \lambda_{nk}$ be eigenvalues of the normalized Laplacian $N = D^{\dagger/2} L D^{\dagger/2}$ of G and let $f_1, \ldots, f_{nk}$ be a corresponding set of orthonormal eigenvectors. For a vertex subset $U \subseteq V$ of size t, we say the vertex-induced subgraph $G[U]$ (α, β, ζ)-preserves G for some $\alpha, \beta \geqslant 1$, $\zeta \in (0, 1)$ iff

1. The null space of $D_{UU}^{\dagger/2} L_{G[U]} D_{UU}^{\dagger/2}$ is exactly the linear span of $\{(f_i)_U : \lambda_i = 0\}$.
2. For all vectors $x \in \mathbb{R}^{|U|k}$ such that $x^T (f_i)_U = 0, \forall i : \lambda_i = 0$,

$$x^T \left(D_{UU}^{\dagger/2} L_{G[U]} D_{UU}^{\dagger/2} \right)^{\dagger} x \leqslant \cdot x^T \left(\alpha \cdot \frac{n^2}{t^2} \sum_{i : \lambda_i \in (0, \zeta]} \frac{1}{\lambda_i} (f_i)_U (f_i)_U^T + \beta \cdot \frac{n}{t} \cdot \frac{1}{\zeta} I \right) x.$$

Therefore, G $(1, 1, \zeta)$-preserves itself for any ζ.

Theorem 4.9.27. *Let* $G = (V, E)$ *be a* $k \times k$ *matrix-weighted* (γ, ζ, ψ)*-almost regular expander with* $\zeta \leqslant 1/\log n$*. For an*

$$s \geqslant 2 \cdot 10^6 \cdot \gamma \psi \zeta^{-1} k^2 n^{\frac{50000}{\log \log n}},$$

let $C \subseteq V$ *be a uniformly random vertex subset of size* s*. Then with probability at least* $1 - n^{-7}$*, the induced subgraph* $G[C]$ $(2, n^{\frac{4096}{\log \log n}}, \zeta)$*-preserves* G*.*

The theorem is a consequence of applying the following lemma $O(\frac{\log n}{\log \log n})$ times.

Lemma 4.9.28. *Let* $G = (V, E)$ *be a* $k \times k$ *matrix-weighted* (γ, ζ, ψ)*-almost regular expander with*

$\zeta \leqslant 1/\log n$. *Suppose there exist an* $\alpha \in [1,2]$, *a* $\beta \geqslant 1$, *and a* $\delta \in (0,1)$, *and a*

$$t \geqslant 2 \cdot 10^6 \cdot \alpha\beta\gamma\psi\zeta^{-1}k^2 \log^{10} n,$$

such that, a subgraph $G[U]$ *induced by a random vertex subset* U *of size* t (α, β, ζ)*-preserves* G *with probability at least* $1 - \delta$.

For an $s \in [t/\log n, t]$, *let* $C \subseteq V$ *be a uniformly random subset of size* s. *Then with probability at least* $1 - \delta - n^{-8}$, *the induced subgraph* $G[C]$ $((1 + \frac{1}{\log n})\alpha, 64\beta, \zeta)$*-preserves* G.

Proof of Lemma 4.9.28. Let $\lambda_1 \leqslant \ldots \leqslant \lambda_{nk}$ be eigenvalues of the normalized Laplacian $N = D^{\dagger/2}LD^{\dagger/2}$ of G and let $f_1, \ldots, f_{nk}$ be a corresponding set of orthonormal eigenvectors. Let z be such that $\lambda_z = 0 < \lambda_{z+1}$.

Consider generating C by first randomly sampling a subset $U \subseteq V$ of size t, and then sub-sampling a subset $C \subseteq U$ of size s from U. We first prove a claim about the Schur complement of $L_{G[U]}$ onto C.

Claim 4.9.29. *With probability at least* $1-\delta-2n^{-10}$, *we have for all* $x \in \mathbb{R}^{|C|k}$ *such that* $x^T(f_i)_C = 0, \forall i : \lambda_i = 0$,

$$x^T \left(D_{CC}^{\dagger/2} \mathrm{SC}\left(L_{G[U]}, C\right) D_{CC}^{\dagger/2} \right)^\dagger x \leqslant x^T \left(\alpha \cdot \frac{n^2}{t^2} \sum_{i:\lambda_i \in (0,\zeta]} \frac{1}{\lambda_i} (f_i)_C (f_i)_C^T + \beta \cdot \frac{16s}{t} \cdot \frac{n}{t} \cdot \frac{1}{\zeta} I \right) x.$$

Proof of Claim 4.9.29. Let $F := U \setminus C$. By Proposition 4.9.22, we can factorize $L_{G[U]}$ as

$$L_{G[U]} = \begin{pmatrix} I_{FF} & 0 \\ (L_{G[U]})_{CF}(L_{G[U]})_{FF}^\dagger & I_{CC} \end{pmatrix} \begin{pmatrix} (L_{G[U]})_{FF} & 0 \\ 0 & \mathrm{SC}(L_{G[U]}, C) \end{pmatrix} \begin{pmatrix} I_{FF} & 0 \\ (L_{G[U]})_{CF}(L_{G[U]})_{FF}^\dagger & I_{CC} \end{pmatrix}^T.$$

By multiplying $D_{UU}^{\dagger/2}$ on both sides, we get a factorization of $D_{UU}^{\dagger/2} L_{G[U]} D_{UU}^{\dagger/2}$:

$$
\begin{aligned}
& D_{UU}^{\dagger/2} L_{G[U]} D_{UU}^{\dagger/2} \\
&= \begin{pmatrix} D_{FF}^{\dagger/2} & 0 \\ D_{CC}^{\dagger/2}(L_{G[U]})_{CF}(L_{G[U]})_{FF}^{\dagger} & D_{CC}^{\dagger/2} \end{pmatrix} \begin{pmatrix} (L_{G[U]})_{FF} & 0 \\ 0 & \mathrm{SC}(L_{G[U]}, C) \end{pmatrix} \\
&\quad \begin{pmatrix} D_{FF}^{\dagger/2} & 0 \\ D_{CC}^{\dagger/2}(L_{G[U]})_{CF}(L_{G[U]})_{FF}^{\dagger} & D_{CC}^{\dagger/2} \end{pmatrix}^{T} \\
&= \begin{pmatrix} I_{FF} & 0 \\ D_{CC}^{\dagger/2}(L_{G[U]})_{CF}(L_{G[U]})_{FF}^{\dagger} D_{FF}^{1/2} & I_{CC} \end{pmatrix} \begin{pmatrix} D_{FF}^{\dagger/2}(L_{G[U]})_{FF} D_{FF}^{\dagger/2} & 0 \\ 0 & D_{CC}^{\dagger/2}\mathrm{SC}(L_{G[U]}, C) D_{CC}^{\dagger/2} \end{pmatrix} \\
&\quad \begin{pmatrix} I_{FF} & 0 \\ D_{CC}^{\dagger/2}(L_{G[U]})_{CF}(L_{G[U]})_{FF}^{\dagger} D_{FF}^{1/2} & I_{CC} \end{pmatrix}^{T},
\end{aligned}
$$

where in the last inequality we have used that the rows of $(L_{G[U]})_{FF}$ are all in the range of D_{FF}. In the case that $G[U]$ (α, β, ζ)-preserves G, we have that the null space of $D_{UU}^{\dagger/2} L_{G[U]} D_{UU}^{\dagger/2}$ is exactly the linear span of $\{(f_i)_U : 1 \leqslant i \leqslant z\}$. Then by Fact 4.9.23,

$$
\left(D_{CC}^{\dagger/2}\mathrm{SC}(L_{G[U]}, C) D_{CC}^{\dagger/2}\right)^{\dagger} = \Pi^{\perp}_{(f_1)_C, \ldots, (f_z)_C} \left(\left(D_{UU}^{\dagger/2} L_{G[U]} D_{UU}^{\dagger/2}\right)^{\dagger}\right)_{CC} \Pi^{\perp}_{(f_1)_C, \ldots, (f_z)_C}.
$$

Let $\tilde{\lambda}_1, \ldots, \tilde{\lambda}_\ell$ be all eigenvalues of $D_{UU}^{\dagger/2} L_{G[U]} D_{UU}^{\dagger/2}$ that are in the range $\left(0, \frac{1}{2} \cdot \frac{t}{n}\right]$. Let $\tilde{f}_1, \ldots, \tilde{f}_\ell$ be a set of orthonormal eigenvectors, and define the embedding $\tilde{F} : V \to R^{\ell \times k}$ by

$$
\tilde{F}(u) = \begin{pmatrix} (\tilde{f}_1)_u^T \\ \vdots \\ (\tilde{f}_\ell)_u^T \end{pmatrix} \in \mathbb{R}^{\ell \times k}.
$$

Since $s \in [t/\log n, t]$ and $\zeta \leqslant 1/\log n$, we have $\frac{t}{2s} \cdot \frac{t}{n} \cdot \zeta \leqslant \frac{1}{2} \cdot \frac{t}{n}$. Then we have

$$\left(D_{UU}^{\dagger/2} L_{G[U]} D_{UU}^{\dagger/2}\right)^{\dagger} \preceq$$
$$\Pi_{(f_1)_U,\ldots,(f_z)_U}^{\perp} \left(\alpha \cdot \frac{n^2}{t^2} \sum_{i:\lambda_i \in (0,\zeta]} \frac{1}{\lambda_i} (f_i)_U (f_i)_U^T + \beta \cdot \frac{4n}{t} \sum_{j=1}^{\ell} \frac{1}{\zeta} \tilde{f}_j \tilde{f}_j^T + \beta \cdot \frac{4}{\frac{t}{2s} \cdot \frac{t}{n} \cdot \zeta} I\right) \Pi_{(f_1)_U,\ldots,(f_z)_U}^{\perp}. \tag{4.68}$$

Using the matrix Chernoff bound and that $t \geqslant 2 \cdot 10^6 \gamma k \log^5 n$, we have with probability $1 - n^{-10}$ that

$$\frac{1}{2} D_{G[U]} \preceq \frac{t}{n} D_{UU} \preceq 2 D_{G[U]}. \tag{4.69}$$

As a result, $G[U]$ is 2γ-almost regular. Therefore by applying Lemma 4.6.4 to $G[U]$ with $\tilde{D} = \frac{t}{n} D_{UU}$ with $\kappa = 2$ and $\delta = \frac{1}{2}$, we have for each vertex u

$$\lambda_{\max}\left(\tilde{F}(u)^T \tilde{F}(u)\right) \leqslant \frac{32\gamma k}{n}.$$

By Lemma 4.9.25, with probability $1 - n^{-10}$,

$$\sum_{j=1}^{\ell} (\tilde{f}_j)_C (\tilde{f}_j)_C^T \preceq \frac{2s}{t} I. \tag{4.70}$$

Now by restricting both sides of (4.68) to vertices in C and then plugging in (4.70), we have our desired claim. □

We now set up a matrix-valued martingale. To that end, we also need the notion of graph squaring as we did in the ordinary graph case.

Definition 4.9.30 (Graph squaring)**.** For a graph $G = (V, E)$, define the square of G by

$$L_{G^2} \stackrel{\text{def}}{=} D - AD^{\dagger}A.$$

The proof of the following claim is deferred to Appendix B.5.

Claim 4.9.31. *$L_{G^2} \preceq 2L_G$ for any graph G.*

Below, we will write $\mathrm{SC}_n(L, C)$ or $\mathrm{SC}_n(G, C)$ to denote an $nk \times nk$ matrix obtained by augmenting $\mathrm{SC}(L, C)$ to $nk \times nk$ by adding zeros on row and column blocks in $V \setminus C$. We will also abuse the notation a bit and write $L_{G[C]}$ to denote an $nk \times nk$ matrix obtained by augmenting $L_{G[C]}$ to $nk \times nk$ by adding zeros on row and column blocks in $V \setminus C$. The proof of the following proposition is also deferred to Appendix B.5.

Proposition 4.9.32. *$\sum_{u \in V} \mathrm{SC}_n(L_G, V \setminus \{u\}) = L_{G^2} + (n-2)L_G$ for any graph G.*

Let $v_1, \ldots, v_t$ be a uniformly random permutation of vertices in U. Define

$$V_i = \{v_1, \ldots, v_i\}.$$

Let $H = G[U]$. Consider the following sequence of matrices:

$$\begin{aligned}
X_0 &= L_H \\
X_1 &= L_H(:, v_1) D_{v_1}^{\dagger} L_H(:, v_1)^T + \left(1 + \frac{2}{t-2}\right) L_{H[U-V_1]} + \frac{1}{t}\left(L_{H^2} - 2L_H\right) \\
X_2 &= L_H(:, v_1) D_{v_1}^{\dagger} L_H(:, v_1)^T + \frac{1}{t}\left(L_{H^2} - 2L_H\right) + \left(1 + \frac{2}{t-2}\right) \cdot \\
&\quad \left(L_{H[U-V_1]}(:, v_2) \left(D_{v_2}^{H[U-V_1]}\right)^{\dagger} L_{H[U-V_1]}(:, v_2)^T + \left(1 + \frac{2}{t-3}\right) L_{H[U-V_2]} \right. \\
&\quad \left. + \frac{1}{t-1}\left(L_{H[U-V_1]^2} - 2L_{H[U-V_1]}\right) \right)
\end{aligned}$$

X_{i+1} is obtained by replacing the $L_{H[U-V_i]}$ in X_i by

$$\begin{aligned}
&L_{H[U-V_i]}(:, v_{i+1}) \left(D_{v_{i+1}}^{H[U-V_i]}\right)^{\dagger} L_{H[U-V_i]}(:, v_{i+1})^T + \left(1 + \frac{2}{t-2-i}\right) L_{H[U-V_{i+1}]} \\
&\quad + \frac{1}{t-i}\left(L_{H[U-V_i]^2} - 2L_{H[U-V_i]}\right). \qquad (4.71)
\end{aligned}$$

Here, we have used D_v^H to denote the degree of vertex v in graph H. The proof of the following

claim is also deferred to Appendix B.5.

Claim 4.9.33. *$X_0, X_1, X_2, \ldots$ is a matrix-valued martingale.*

Define $Y_i = L_H^{\dagger/2} X_i L_H^{\dagger/2} - L_H^{\dagger/2} L_H L_H^{\dagger/2}$. As $X_0, X_1, \ldots$ is a martingale, so is $Y_0, Y_1, \ldots$, by the linearity of expectation. Additionally, since $X_0 = L_H$, $Y_0, Y_1, \ldots$ has zero expectation. We focus on the first $t - s + 1$ terms $Y_0, Y_1, Y_2, \ldots, Y_{t-s}$. First we calculate the difference sequence $Z_i = Y_{i+1} - Y_i$:

$$\begin{aligned} Z_i = &\prod_{j=1}^{i} \left(1 + \frac{2}{t-1-j}\right) \cdot \\ &L_H^{\dagger/2} \Bigg(- \mathrm{SC}_n\left(H[U - V_i], U - V_{i+1}\right) + \left(1 + \frac{2}{t-2-i}\right) L_{H[U-V_{i+1}]} \\ &\quad - \frac{1}{t-i}\left(L_{H[U-V_i]^2} - 2L_{H[U-V_i]}\right) \Bigg) L_H^{\dagger/2}, \end{aligned}$$

where we have used the definition of Schur complements. We then consider to bound the maximum (in absolute value) eigenvalue of Z_i. For the last term, we have

$$\lambda_{\max}\left(L_H^{\dagger/2} \frac{1}{t-i}\left(L_{H[U-V_i]^2} - 2L_{H[U-V_i]}\right) L_H^{\dagger/2}\right) \leqslant \frac{4}{t-i} \leqslant \frac{4}{s} \leqslant \frac{4\log n}{t} \leqslant \frac{4}{10^6 \log^6 n}, \tag{4.72}$$

where the first inequality follows from $L_{H[U-V_i]^2} \preceq 2L_{H[U-V_i]} \preceq 2L_H$ by Claim 4.9.31. Now to bound the first two terms, we first rearrange them as

$$\left(-\mathrm{SC}_n\left(H[U - V_i], U - V_{i+1}\right) + L_{H[U-V_{i+1}]}\right) + \frac{2}{t-2-i} L_{H[U-V_{i+1}]}. \tag{4.73}$$

Then for the second term of (4.73), we have $L_{H[U-V_{i+1}]} \preceq L_H$, and hence

$$\lambda_{\max}\left(L_H^{\dagger/2} \frac{2}{t-2-i} L_{H[U-V_{i+1}]} L_H^{\dagger/2}\right) \leqslant \frac{2}{t-2-i} \leqslant \frac{2}{s-2} \leqslant \frac{4}{s} \leqslant \frac{4\log n}{t} \leqslant \frac{4}{10^6 \log^6 n}. \tag{4.74}$$

We then claim that the first term of (4.73) can be bounded by

$$\lambda_{\max}\left(L_H^{\dagger/2}\left(\mathrm{SC}_n\left(H[U - V_i], U - V_{i+1}\right) - L_{H[U-V_{i+1}]}\right) L_H^{\dagger/2}\right) \leqslant \alpha \frac{\psi k^2}{t} + \beta \frac{\gamma k}{\zeta t} \leqslant \frac{1}{8 \cdot 10^6 \log^6 n}. \tag{4.75}$$

To see why this is the case, let us define an edge set $E_{i+1} = \{(v_{i+1}, w) \in E : w \in U - V_{i+1}\}$, and write $L_{E_{i+1}}$ to denote the Laplacian of the subgraph of H induced by E_{i+1}. Notice that

$$L_{E_{i+1}}(:, v_{i+1}) = L_{H[U-V_i]}(:, v_{i+1}).$$

Therefore, we have

$$\mathrm{SC}\left(H[U - V_i], U - V_{i+1}\right) - L_{H[U-V_{i+1}]} = \mathrm{SC}\left(L_{E_{i+1}}, U - V_{i+1}\right).$$

Therefore to prove (4.75), it suffices to show that for any vector x such that for all x such that $x \in \mathrm{range}(\mathrm{SC}_n(L_{E_{i+1}}, U - V_{i+1}))$, we have

$$x^T L_H^{\dagger} x \leqslant \left(\alpha \frac{\psi k^2}{t} + \beta \frac{\gamma k}{\zeta t}\right) \cdot x^T \mathrm{SC}\left(L_{E_{i+1}}, U - V_{i+1}\right)^{\dagger} x = \left(\alpha \frac{\psi k^2}{t} + \beta \frac{\gamma k}{\zeta t}\right) \cdot x^T L_{E_{i+1}}^{\dagger} x. \quad (4.76)$$

Since $x \in \mathrm{range}(\mathrm{SC}_n(L_{E_{i+1}}, U - V_{i+1}))$, we can write x as a liner combination of the incidence vectors of the edges in E_{i+1}:

$$x = \sum_{e \in E_{i+1}} c_e b_e$$

for some coefficients c_e's, where we have $x_{v_{i+1}} = \sum_{e \in E_{i+1}} c_e (b_e)_{v_{i+1}} = 0$. Then since the induced subgraph $G[E_{i+1}]$ is a star graph, we have

$$x^T L_{E_{i+1}}^{\dagger} x = \sum_{e \in E_{i+1}} c_e^2.$$

We now calculate $x^T L_H^\dagger x$ by

$$\begin{aligned}
x^T L_H^\dagger x =& x^T D_{UU}^{\dagger/2} \left(D_{UU}^{\dagger/2} L_H D_{UU}^{\dagger/2} \right)^\dagger D_{UU}^{\dagger/2} x \\
\leqslant& x^T D_{UU}^{\dagger/2} \left(\alpha \cdot \frac{n^2}{t^2} \sum_{i:\lambda_i \in (0,\zeta]} \frac{1}{\lambda_i} (f_i)_U (f_i)_U^T + \beta \cdot \frac{n}{t} \cdot \frac{1}{\zeta} I \right) D_{UU}^{\dagger/2} x \\
& \qquad \text{(since } H = G[U] \text{ } (\alpha, \beta, \zeta)\text{-preserves } G) \\
\leqslant& |E_{i+1}| \cdot \alpha \cdot \frac{n^2}{t^2} \sum_{e \in E_{i+1}} c_e^2 \frac{\psi k^2}{n^2} + \beta \frac{n}{t} \cdot \frac{1}{\zeta} \left\| \sum_{e \in E_{i+1}} c_e D_{UU}^{\dagger/2} b_e \right\|^2 \\
& \qquad \text{(by expanding and that } G \text{ is a } (\gamma, \zeta, \psi)\text{-almost regular expander)} \\
\leqslant& \alpha \frac{\psi k^2}{t} \left(\sum_{e \in E_{i+1}} c_e^2 \right) + \beta \frac{n}{t} \cdot \frac{1}{\zeta} \left(\sum_{(w, v_{i+1}) \in E_{i+1}} c_e^2 \left\| D_w^{\dagger/2} \phi_{v_{i+1} w} \right\|^2 \right) \\
& \qquad \text{(by } |E_{i+1}| \leqslant t \text{ and that } \sum_{e \in E_{i+1}} c_e D_{UU}^{\dagger/2} b_e \text{ is zero on } v_{i+1}) \\
\leqslant& \left(\alpha \frac{\psi k^2}{t} + \beta \frac{\gamma k}{\zeta t} \right) \left(\sum_{e \in E_{i+1}} c_e^2 \right) \\
& \qquad \text{(by } \gamma\text{-almost regularity of } G).
\end{aligned}$$

This proves our desired inequality (4.76).

We also note that

$$\prod_{j=1}^{i} \left(1 + \frac{2}{t-1-j} \right) = \frac{t}{t-2} \frac{t-1}{t-3} \cdots \frac{t+1-i}{t-1-i} = \frac{t(t-1)}{(t-i)(t-1-i)} \leqslant \left(\frac{t}{s} \right)^2 \leqslant \log^2 n.$$

Therefore

$$\lambda_{\max}(Z_i) \leqslant \frac{1}{1000 \cdot \log^4 n}. \tag{4.77}$$

Fixing a j, we then consider bounding the spectral norm of $\mathbb{E}\left[Z_j^2 \mid Y_1, \ldots, Y_{j-1}\right]$.

$$
\begin{aligned}
&\mathbb{E}\left[Z_j^2 \mid Y_1, \ldots, Y_{j-1}\right] \\
\preceq &\frac{2}{t-j} \cdot \left(\frac{t}{t-1-j}\right)^4 \sum_{v \in U - V_j} \left(L_H^{\dagger/2}\left(\mathrm{SC}_n(H[U-V_j], U-V_j-\{v\}) - L_{H[U-V_j-\{v\}]}\right) L_H^{\dagger/2}\right)^2 + \\
&\frac{2}{t-j} \cdot \left(\frac{t}{t-1-j}\right)^4 \sum_{v \in U - V_j} \\
&\left(L_H^{\dagger/2}\left(\frac{2}{t-2-j} L_{H[U-V_j-\{v\}]} + \frac{1}{t-j}\left(L_{H[U-V_j]^2} + 2L_{H[U-V_j]}\right)\right) L_H^{\dagger/2}\right)^2 \\
\preceq &\frac{2\log^5 n}{t-j} \cdot \left(\alpha \frac{\psi k^2}{t} + \beta \frac{\gamma k}{\zeta t}\right) \\
&\sum_{v \in U - V_j} L_H^{\dagger/2}\left(\mathrm{SC}_n(H[U-V_j], U-V_j-\{v\}) - L_{H[U-V_j-\{v\}]}\right) L_H^{\dagger/2} + \\
&\frac{2\log^5 n}{t-j} \cdot |U - V_j| \cdot \left(\frac{8}{s}\right)^2 I_{tk \times tk} \qquad \text{(by (4.75),(4.74),(4.72))} \\
\preceq &\frac{2\log^6 n}{t}\left(\alpha \frac{\psi k^2}{t} + \beta \frac{\gamma k}{\zeta t}\right) L_H^{\dagger/2} L_{H[U-V_j]^2} L_H^{\dagger/2} + \frac{10000 \log^7 n}{t^2} I_{tk \times tk} \\
&\qquad \text{(by Proposition 4.9.32 and } t - j \geqslant s \geqslant t/\log n) \\
\preceq &\frac{64}{2 \cdot 10000 \cdot t \log^4 n} I_{tk \times tk} + \frac{10000 \log^7 n}{t^2} I_{tk \times tk} \qquad \text{(by Claim 4.9.31 and the values of } \gamma, \zeta, \psi) \\
\preceq &\frac{1}{1000 t \log^4 n} I_{tk \times tk}.
\end{aligned}
$$

Therefore we have

$$
\lambda_{\max}\left(\mathbb{E}\left[Z_j^2 \mid Y_1, \ldots, Y_{j-1}\right]\right) \leqslant \frac{1}{1000 t \log^4 n}
$$

and

$$
\begin{aligned}
\lambda_{\max}\left(\sum_{j=0}^{t-s-1} \mathbb{E}\left[Z_j^2 \mid Y_1, \ldots, Y_{j-1}\right]\right) &\leqslant \sum_{j=0}^{t-s-1} \lambda_{\max}\left(\mathbb{E}\left[Z_j^2 \mid Y_1, \ldots, Y_{j-1}\right]\right) \\
&\leqslant t \cdot \frac{1}{1000 t \log^4 n} \\
&= \frac{1}{1000 \log^4 n}. \qquad (4.78)
\end{aligned}
$$

We now invoke Theorem 4.9.19 with $R = \frac{1}{1000\log^4 n}$, $\sigma^2 = \frac{1}{1000\log^4 n}$, and $t = \frac{1}{10\log n}$, and get

$$\Pr\left[\exists i \in [0, n-s] : \lambda_{\max}(Y_i) \geqslant 1/2\right] \leqslant n \cdot \exp\left(-\frac{\frac{1}{200\log^2 n}}{\frac{1}{1000\log^4 n} + \frac{1}{30000\log^5 n}}\right) \leqslant n^{-10}.$$

This coupled with Fact 4.9.23 implies that

$$L_{G[C]} \succeq \left(1 - \frac{1}{5\log n}\right) \cdot \frac{s^2}{t^2} \cdot \mathrm{SC}(L_{G[U]}, C). \tag{4.79}$$

Then using Claim 4.9.29, for all $x \in \mathbb{R}^{|C|k}$ such that $x^T(f_i)_C = 0, \forall i : \lambda_i = 0$,

$$\begin{aligned}
&x^T \left(D_{CC}^{\dagger/2} L_{G[C]} D_{CC}^{\dagger/2}\right)^{\dagger} x && (4.80)\\
\leqslant &\left(1 + \frac{1}{2\log n}\right) \cdot \frac{t^2}{s^2} \cdot x^T \left(\alpha \cdot \frac{n^2}{t^2} \sum_{i:\lambda_i \in (0,\zeta]} \frac{1}{\lambda_i} (f_i)_C (f_i)_C^T + \beta \cdot \frac{16s}{t} \cdot \frac{n}{t} \cdot \frac{1}{\zeta} I\right) x \\
\leqslant &x^T \left(\left(1 + \frac{1}{\log n}\right) \alpha \cdot \frac{n^2}{s^2} \sum_{i:\lambda_i \in (0,\zeta]} \frac{1}{\lambda_i} (f_i)_C (f_i)_C^T + 64\beta \cdot \frac{n}{s} \cdot \frac{1}{\zeta} I\right) x. && (4.81)
\end{aligned}$$

(4.79),(4.81) together imply that $G[C]$ $((1 + \frac{1}{\log n})\alpha, 64\beta, \zeta)$-preserves G, as desired. □

4.10. A lower bound for weighted spectral sparsification

In this section, we prove a superlinear lower bound of $n^{21/20-o(1)}$ for computing some $O(1)$-spectral sparsifier.

Theorem 4.10.1 (Lower bound for weighted spectral sparsification)**.** *There exist constants $\varepsilon, \delta \in (0,1)$ such that any incidence sketch of N measurements that computes a $(1+\varepsilon)$-spectral sparsifier with probability $\geqslant 1-\delta$ on any w must satisfy $N \geqslant n^{21/20-o(1)}$.*

By Proposition 4.2.3, in order to prove Theorem 4.10.1, it suffices to prove Theorem 4.2.9. Recall that $B^w \in \mathbb{R}^{\binom{n}{2} \times n}$ is the weighted signed edge-vertex incidence matrix of the input graph generated from the distribution specified in Section 4.2.

Theorem 4.2.9. *For any fixed sketching matrix $\Phi \in \mathbb{R}^{k \times \binom{n}{2}}$ where $k \leqslant n^{1/20-\varepsilon}$ for some constant*

$\varepsilon > 0$, *we have*

$$\mathbb{E}_\pi \left[d_{\mathrm{TV}} \left((\Phi B^w)_{\pi,\mathrm{yes}}, (\Phi B^w)_{\pi,\mathrm{no}} \right) \right] \leqslant o(1).$$

Proof of Theorem 4.2.9. Similar to Section 4.2.3, we can assume w.l.o.g. the number of $(u, v) \in E$ for which $\phi_{uv} \neq 0$ is at least $\Omega(n^2)$, since otherwise

$$\mathbb{E}_\pi \left[d_{\mathrm{TV}} \left((\Phi B^w)_{\pi,\mathrm{yes}}, (\Phi B^w)_{\pi,\mathrm{no}} \right) \right] \leqslant O(1) \cdot \Pr_\pi \left[\phi_{\pi(1)\pi(n/2+1)} \neq 0 \right] \leqslant o(1),$$

and we would already have our desired result. Let us then define the $k \times k$ matrix-weighted graph $H_\phi = (V, E_\phi)$, where E_ϕ contains *all* edges (u, v) whose $\phi_{uv} \neq 0$ (including the ones not present in the input graph), and each edge (u, v) has matrix weight $\phi_{uv}\phi_{uv}^T$. Thus by the above assumption we have $|E_\phi| \geqslant \Omega(n^2)$.

By Theorem 4.2.19, there exists a scaling $s : E_\phi \to [0, 1]$ s.t. H_ϕ^s is a (γ, ζ, ψ)-almost regular expander, where $(\gamma, \zeta, \psi) = (8 \log n, 1/\log n, 16k^2 \log^3 n)$, and $|(u, v) \in E_\phi : s_{uv} < 1| \leqslant o(n^2)$. Let $\lambda_1 \leqslant \ldots \leqslant \lambda_{nk}$ be the eigenvalues of the normalized Laplacian matrix of H_ϕ^s, and let $f_1, \ldots, f_{nk} \in \mathbb{R}^{nk}$ be a corresponding set of orthonormal eigenvectors.

For $i = 0, \ldots, n^{4/5} - 1$, let B_i denote the vertices in the i^{th} block of the input graph:

$$B_i \stackrel{\text{def}}{=} \left\{ \pi(n^{1/5} i + 1), \ldots, \pi(n^{1/5} i + n^{1/5}) \right\}.$$

Since $k \leqslant n^{1/20-\varepsilon}$, we have $2 \cdot 10^6 \gamma \psi \zeta^{-1} k^2 n^\varepsilon \leqslant n^{1/5}$. Thus by Theorem 4.2.20, with probability $1 - 1/n^4$, all vertex-induced subgraphs $H_\phi^s[B_i \cup B_{i+1}]$ preserve H_ϕ^s in the sense of (4.7). Using this fact, we prove the following claim:

Claim 4.10.2. *For each edge (x, y) with $s_{xy} = 1$, conditioned on (x, y) being the crossing edge (i.e. $\{\pi(1), \pi(n/2 + 1)\} = \{x, y\}$), with probability at least $1 - 1/n^2$ over π, we have that b_{xy} is in the*

range of L_π, and

$$b_{xy}^T L_\pi^\dagger b_{xy} \leqslant \frac{k^2}{n^{2/5-o(1)}}. \tag{4.82}$$

Once again by $k \leqslant n^{1/20-\varepsilon}$, the RHS of (4.82) is $o(1)$. Since $s_{uv} = 1$ holds for a $1 - o(1)$ fraction of the edges, we have, by Proposition 4.2.4, $\mathbb{E}_\pi \left[d_{\mathrm{TV}} \left((\Phi B^w)_{\pi,\mathrm{yes}}, (\Phi B^w)_{\pi,\mathrm{no}} \right) \right] = o(1)$. □

Proof of Claim 4.10.2. Consider drawing π from the conditional distribution on $\pi(1) = x$ and $\pi(n/2+1) = y$. Suppose w.l.o.g. (x, y) is oriented $x \to y$, so $(b_{xy})_x = \phi_{xy}$ and $(b_{xy})_y = -\phi_{xy}$. Let E_π be the set of edges of $\mathcal{H}_\pi$, where we recall that $\mathcal{H}_\pi$ consists of the non-crossing edges, such that each edge (u, v) has matrix weight $n^{4/5} \log^{-1} n \phi_{uv} \phi_{uv}^T \in \mathbb{R}^{k \times k}$. Let us call a function $\mathbf{f} : E_\pi \to \mathbb{R}$ a *flow* in the graph $\mathcal{H}_\pi$, and say $\mathbf{f}$ routes a *demand* $d \in \mathbb{R}^{nk}$ if $\sum_{e \in E_\pi} \mathbf{f}_e b_e = d$.

By Fact 2.3.6, to prove (4.82), it suffices to show that there exists a flow $\mathbf{f}$ in $\mathcal{H}_\pi$ that routes demand $n^{-2/5} b_{xy}$ satisfying $\|\mathbf{f}\|_2^2 \leqslant \frac{k^2}{n^{2/5-o(1)}}$. By scaling, it then suffices to show that there is a flow $\mathbf{f}$ that routes demand b_{xy} satisfying $\|\mathbf{f}\|_2^2 \leqslant k^2 n^{2/5+o(1)}$. To construct such a flow, we consider first breaking the demand b_{xy} into demands $d_0, d_1, \ldots, d_{n^{4/5}/2-1} \in \mathbb{R}^{nk}$ such that

1. $\sum_{0 \leqslant i < n^{4/5}/2} d_i = b_{xy}$.
2. d_i is supported on $B_i \cup B_{i+1}$.

We will then route each d_i in the vertex induced subgraph $H_\phi[B_i \cup B_{i+1}]$ using a flow $\mathbf{f}_i$, and finally obtain a flow $\mathbf{f} = \sum_{0 \leqslant i < n^{4/5}/2} \mathbf{f}_i$ that routes b_{xy} and satisfies $\|\mathbf{f}\|_2^2 \leqslant 2 \sum_{0 \leqslant i < n^{4/5}/2} \|\mathbf{f}_i\|_2^2$.

To show how these d_i's are constructed, let us first define vectors $\tilde{d}_0, \ldots, \tilde{d}_{n^{4/5}/2} \in \mathbb{R}^{nk}$ where $\tilde{d}_i$ is supported on B_i. We will later define each d_i by letting $d_i = \tilde{d}_i - \tilde{d}_{i+1}$. Let $D \in \mathbb{R}^{nk \times nk}$ be the degree matrix of H_ϕ^s. For now we would like $\tilde{d}_i$ to satisfy

$$\forall j : 0 \leqslant \lambda_j \leqslant \zeta, \quad f_j^T D^{\dagger/2} \tilde{d}_i = (1 - \mu_i)(f_j)_x^T D_{xx}^{\dagger/2} \phi_{xy} + \mu_i (f_j)_y^T D_{yy}^{\dagger/2} \phi_{xy}. \tag{4.83}$$

where we define

$$\mu_i := \frac{i}{n^{4/5}/2}.$$

In particular, we would like $f_j^T D^{\dagger/2} \tilde{d}_0 = (f_j)_x^T D_{xx}^{\dagger/2} \phi_{xy}$ and $f_j^T D^{\dagger/2} \tilde{d}_{n^{4/5}/2} = (f_j)_y^T D_{yy}^{\dagger/2} \phi_{xy}$ for all j such that $0 \leqslant \lambda_j \leqslant \zeta$. Thus, we let $\tilde{d}_0$ be such that

$$(\tilde{d}_0)_u = \begin{cases} \phi_{xy} & u = x \\ 0 & \text{otherwise} \end{cases}$$

and let $\tilde{d}_{n^{4/5}/2}$ be such that

$$(\tilde{d}_{n^{4/5}/2})_u = \begin{cases} \phi_{xy} & u = y \\ 0 & \text{otherwise.} \end{cases}$$

We would then like to construct the remaining $\tilde{d}_i$'s such that each $\left\| D^{\dagger/2} \tilde{d}_i \right\|_2$ is small. First let us write the linear equations (4.83) in matrix form. Let ℓ be s.t. $\lambda_\ell \leqslant \zeta < \lambda_{\ell+1}$. Define a function $F : V \to \mathbb{R}^{\ell \times k}$ such that

$$F(u) = \begin{pmatrix} (f_1)_u^T \\ \vdots \\ (f_\ell)_u^T \end{pmatrix}$$

and define another function $G : \left\{ B_0, \ldots, B_{n^{4/5}/2} \right\} \to \mathbb{R}^{\ell \times n^{1/5} k}$ such that

$$G(B_i) = \begin{pmatrix} (f_1)_{B_i}^T \\ \vdots \\ (f_\ell)_{B_i}^T \end{pmatrix}$$

Then we can write (4.83) as

$$G(B_i)D^{\dagger/2}_{B_iB_i}(\tilde{d}_i)_{B_i} = (1-\mu_i)F(x)D^{\dagger/2}_{xx}\phi_{xy} + \mu_i F(y)D^{\dagger/2}_{yy}\phi_{xy} \tag{4.84}$$

Let us write the RHS as $r_i := (1-\mu_i)F(x)D^{\dagger/2}_{xx}\phi_{xy} + \mu_i F(y)D^{\dagger/2}_{yy}\phi_{xy}$. Then by Fact 2.3.6, we can find a $(\tilde{d}_i)_{B_i}$ satisfying (4.84) such that $\left\|D^{\dagger/2}_{B_iB_i}(\tilde{d}_i)_{B_i}\right\|_2^2 = r_i^T(G(B_i)G(B_i)^T)^{-1}r_i$. Now notice

$$G(B_i)G(B_i)^T = \sum_{u\in B_i} F(u)F(u)^T.$$

Since $f_1,\ldots,f_\ell$ are orthonormal, we have $\sum_{u\in V} F(u)F(u)^T = I_{\ell\times\ell}$. By Lemma 4.6.4, we have for all $u\in V$ that $\lambda_{\max}(F(u)F(u)^T) \leqslant \frac{\gamma k}{(1-\zeta)^2 n} \leqslant \frac{16k\log n}{n}$. Since the vertices in B_i are chosen uniformly at random, by a Matrix Chernoff bound (Theorem 2.3.4), we have with probability $\geqslant 1-n^{-5}$ that $\frac{1}{2}I \preceq n^{4/5}G(B_i)G(B_i)^T \preceq 2I$, which implies $r_i^T(G(B_i)G(B_i)^T)^{-1}r_i \leqslant 2n^{4/5}\left\|r_i\right\|^2$. As a result, we can find $\tilde{d}_1,\ldots,\tilde{d}_{n^{4/5}/2-1}$ such that

$$\left\|D^{\dagger/2}\tilde{d}_i\right\|_2^2 \leqslant 4n^{4/5}\left(\left\|F(x)D^{\dagger/2}_{xx}\phi_{xy}\right\|_2^2 + \left\|F(y)D^{\dagger/2}_{yy}\phi_{xy}\right\|_2^2\right).$$

For simplicity let $C_i := B_i\cup B_{i+1}$. We then construct each d_i for $0\leqslant i < n^{4/5}/2$ by letting $d_i := \tilde{d}_i - \tilde{d}_{i+1}$. Thus d_i is indeed supported on C_i, and we have $\sum_{0\leqslant i<n^{4/5}/2} d_i = b_{xy}$. Now we consider how to route each d_i in $H_\phi[B_i\cup B_{i+1}]$. Once again by Fact 2.3.6, there is a flow $\mathbf{f}_i$ with

ℓ_2^2-norm $(d_i)_{C_i}^T L^\dagger_{H_\phi[C_i]}(d_i)_{C_i} \leqslant (d_i)_{C_i}^T L^\dagger_{H^s_\phi[C_i]}(d_i)_{C_i}$. We then write

$$\begin{aligned}
&(d_i)_{C_i}^T L^\dagger_{H^s_\phi[C_i]}(d_i)_{C_i}\\
=&(d_i)_{C_i}^T D^{\dagger/2}_{C_iC_i}\left(D^{\dagger/2}_{C_iC_i} L_{H^s_\phi[C_i]} D^{\dagger/2}_{C_iC_i}\right)^\dagger D^{\dagger/2}_{C_iC_i}(d_i)_{C_i}\\
\leqslant& n^{o(1)}(d_i)_{C_i}^T D^{\dagger/2}_{C_iC_i}\left(\frac{n^2}{n^{2/5}}\sum_{j:\lambda_j\in(0,\zeta]}\frac{1}{\lambda_j}(f_j)_{C_i}(f_j)_{C_i}^T + \frac{n}{n^{1/5}}(\log n)I\right) D^{\dagger/2}_{C_iC_i}(d_i)_{C_i}\\
\leqslant& n^{8/5+o(1)}\sum_{j:\lambda_j\in(0,\zeta]}\frac{1}{\lambda_j}\left(f_j^T D^{\dagger/2}(\tilde{d}_i-\tilde{d}_{i+1})\right)^2 + n^{4/5+o(1)} d_i^T D^\dagger d_i\\
=& n^{8/5+o(1)}\sum_{j:\lambda_j\in(0,\zeta]}\frac{1}{\lambda_j}\left(\frac{1}{n^{4/5}/2} f_j^T D^{\dagger/2} b_{xy}\right)^2 + n^{4/5+o(1)}\left(\tilde{d}_i^T D^\dagger \tilde{d}_i + \tilde{d}_{i+1}^T D^\dagger \tilde{d}_{i+1}\right)\\
=& n^{o(1)} b_{xy}^T D^{\dagger/2}\left(\sum_{j:\lambda_j\in(0,\zeta]}\frac{1}{\lambda_j} f_j f_j^T\right) D^{\dagger/2} b_{xy} + n^{4/5+o(1)}\left(\tilde{d}_i^T D^\dagger \tilde{d}_i + \tilde{d}_{i+1}^T D^\dagger \tilde{d}_{i+1}\right).
\end{aligned}$$

Notice that the first term in the above is at most $\frac{n^{o(1)}\psi k^2}{n^2} \leqslant \frac{n^{o(1)}k^4}{n^2}$ by the expander property and $s_{xy}=1$. The second term for each $0<i<n^{4/5}/2-1$ is at most

$$n^{8/5+o(1)}\left(\left\|F(x)D^{\dagger/2}_{xx}\phi_{xy}\right\|_2^2 + \left\|F(y)D^{\dagger/2}_{yy}\phi_{xy}\right\|_2^2\right) \leqslant \frac{k^2}{n^{2/5-o(1)}},$$

where the inequality follows from γ-regularity and that $\lambda_{\max}(F(u)^T F(u)) \leqslant \frac{16k\log n}{n}$. For $i=0$ or $i=n^{4/5}/2-1$, the second term is at most

$$\begin{aligned}
n^{4/5+o(1)}\left(\left\|D^{\dagger/2}_{xx}\phi_{xy}\right\|_2^2 + \left\|D^{\dagger/2}_{yy}\phi_{xy}\right\|_2^2\right) +&\\
n^{8/5+o(1)}\left(\left\|F(x)D^{\dagger/2}_{xx}\phi_{xy}\right\|_2^2 + \left\|F(y)D^{\dagger/2}_{yy}\phi_{xy}\right\|_2^2\right) &\leqslant \frac{k}{n^{1/5-o(1)}}.
\end{aligned}$$

Finally by letting $\mathbf{f}=\sum_{0\leqslant i<n^{4/5}/2}\mathbf{f}_i$, we have $\|\mathbf{f}\|_2^2 \leqslant k^2 n^{2/5+o(1)}$ as desired. □

CHAPTER 5

A PARALLEL APPROXIMATE MAXIMUM FLOW ALGORITHM IN NEAR-LINEAR WORK AND POLY-LOGARITHMIC DEPTH

In this chapter, we present our parallel approximate maximum flow algorithm with near-linear work and poly-logarithmic depth. The results from this chapter are based on my work [AKL$^+$24] joint with Agarwal, Khanna, Patil, Wang, White, and Zhong.

We start with an overview of our algorithm in Section 5.1. Then we present the first key step of our algorithm, namely a new construction of a flow sparsifier called *congestion approximator*, in Section 5.2 - Section 5.4. In what follows, we present the second key step of our algorithm, namely a parallel flow decomposition algorithm, in Section 5.5. Finally, we present the applications of our parallel maximum flow algorithm to the parallel computation of (balanced) sparsest cuts, hierarchical clustering, fair cuts, and approximate Gomory-Hu trees.

5.1. Overview of the Algorithm

The starting point of our discussion is Sherman's framework [She13b, She17b], which was initially proposed for computing fast approximate undirected max-flows in the *sequential* setting. Given the optimization problem defined in Equation 1.1, let $\mathsf{opt}_G(b) := \min_{f:Bf=b} \|C^{-1}f\|_\infty$ be the minimum congestion of any feasible flow routing given demands b in the graph G. Sherman showed that if we have a matrix Φ such that for *any* demand vector b, $\|\Phi b\|_\infty$ is always an α-approximation to $\mathsf{opt}(b)$, i.e., $\|\Phi b\|_\infty \leqslant \mathsf{opt}(b) \leqslant \alpha \cdot \|\Phi b\|_\infty$, then there is an iterative algorithm to compute a $(1+\varepsilon)$-approximate minimum congestion flow. The matrix Φ is called a *α-congestion approximator matrix*, and the algorithm requires $\mathrm{poly}(\alpha, \varepsilon^{-1}, \log n)$ iterations[11] where each iteration requires in total $O(m)$-time algorithmic operations plus computing the matrix-vector products Φx and $\Phi^T y$ for some arbitrary vectors x and y.

A line of work in the parallel transshipment (ℓ_1 flow) literature [ASZ20, Li20, RGH$^+$22, ZGY$^+$22]

[11] To be precise, Sherman's algorithm upon termination routes a flow that *almost* satisfies the desired demand b, and the negligible residual demand is such that it can be routed in the flow network with $1/\mathrm{poly}(m)$ congestion. We get a feasible flow by trivially routing this residual demand along a maximum spanning tree, which is known to admit an efficient parallel construction with $\tilde{O}(m)$ work and $\mathrm{polylog}(n)$ depth [PR02].

observed that Sherman's framework can be adapted to the *parallel* computing regime. In particular, they showed the following: given an α-congestion approximator matrix Φ, there is a PRAM algorithm which outputs a $(1+\varepsilon)$-approximate minimum congestion flow. The depth of the algorithm is $\text{poly}(\alpha, \varepsilon^{-1}, \log n) \cdot (\text{depth}(\Phi x) + \text{depth}(\Phi^T y))$, and the work is $\text{poly}(\alpha, \varepsilon^{-1}, \log n) \cdot (\text{work}(\Phi x) + \text{work}(\Phi^T y) + O(m))$, where $\text{depth}(\cdot), \text{work}(\cdot)$ denote the depth and work required to compute the corresponding sub-problem respectively, and $\Phi x, \Phi^T y$ denote the sub-problems of multiplication between Φ and an arbitrary vector and the multiplication between Φ^T and an arbitrary vector respectively. Therefore, the problem effectively reduces to efficiently computing a good congestion approximator matrix Φ that also allows efficient computation of Φx and $\Phi^T y$.

Räcke, Shah, and Täubig [RST14] gave an efficient algorithm for building a *tree-structured* congestion approximator. Specifically, they showed that given any graph $G = (V, E, c)$, one can construct an $O(\log n)$-depth tree $R = (V_R, E_R, c_R)$ supported on $V_R \supseteq V$ such that for any demand vector b, $\mathsf{opt}_R(b) \leqslant \mathsf{opt}_G(b) \leqslant O(\log^4 n) \cdot \mathsf{opt}_R(b)$. A useful property of the tree R is that the flow that goes from u to the parent of u is equal to the total demand within the subtree of u. This implies that we can write $\mathsf{opt}_R(b) = \|\Phi b\|_\infty$ for a matrix Φ, where each row of Φ corresponds to a node in R, such that the u-th entry of Φb equals the congestion of the tree edge connecting u to its parent:

$$(\Phi b)_u = \frac{\sum_{v \text{ in the subtree of } u} b_v}{c_R(u, \text{parent of } u)}.$$

Furthermore, computing Φx is equivalent to computing the sum of rescaled node weights in each subtree, and computing $\Phi^T y$ is equivalent to computing the sum of rescaled edge weights of the path from each node to the root. Thus, both computations of Φx and $\Phi^T y$ can be done in $\text{polylog}(n)$ depth and $n \cdot \text{polylog}(n)$ work using standard dynamic programming ideas. Hence, our goal becomes to compute such a congestion approximator tree (or hereafter simply *congestion approximator*) R.

The algorithm of [RST14] constructs such a tree by performing a hierarchical decomposition of the graph. At a high level, starting with the vertex set V, they recursively apply a subroutine to partition a given cluster S of vertices into smaller sub-clusters such that the *boundary edges* leaving each sub-cluster are *well-linked*, in the sense that one can route a product multicommodity flow

between them with low congestion. However, this partitioning routine itself requires computing $(1+\varepsilon)$-approximate minimum congestion flows on the induced subgraph $G[S]$, which is the exact same problem (though on a subgraph) we set out to solve by constructing a congestion approximator!

In the *sequential setting*, this chicken-and-egg situation was resolved by Peng [Pen16] using a clever recursive construction, which we summarize below:

1. Given an input graph $G = (V, E, c)$, the goal is to output an $O(\log^4 n)$-congestion approximator tree $R = (V_R, E_R, c_R)$.
2. Simulate the algorithm of [RST14] on G. **Every time** a $(1+\varepsilon)$-approximate minimum congestion flow computation is required on some **vertex-induced subgraph** $G[S]$ of G, do the following:
   (a) Compute a sparsifier H of $G[S]$ such that for any demand b, $\mathsf{opt}_H(b) \leqslant \mathsf{opt}_{G[S]}(b) \leqslant \text{polylog}(n) \cdot \mathsf{opt}_H(b)$. In addition, H can be decomposed into a core graph C and a forest F such that (i) C only has $|S|/\operatorname{polylog}(n)$ nodes and edges, and (ii) each connected component of F has exactly one vertex in C.
   (b) **Recursively** compute an $O(\log^4 n)$-congestion approximator tree R_C for the core graph C.
   (c) Let $R_{G[S]} = R_C \cup F$. It is easy to show that by composing, $R_{G[S]}$ is a $\text{polylog}(n)$-congestion approximator tree for $G[S]$.
   (d) Compute a $(1+\varepsilon)$-approximate minimum congestion flow on $G[S]$ by plugging $R_{G[S]}$ into Sherman's algorithm.
3. Output an $O(\log^4 n)$-congestion approximator tree R for G obtained from simulating the algorithm of [RST14].

Note that since C has significantly smaller size than $G[S]$, the computation of a congestion approximator tree for C is much faster than for $G[S]$. Leveraging this observation, and using a large enough $\text{polylog}(n)$ factor in the size reduction of the sparsifier, [Pen16] showed that the overall running time of their algorithm can be bounded by $m \cdot \text{poly}(\varepsilon^{-1}, \log(n))$.

However, there are two major challenges to convert Peng's algorithm into a small depth PRAM al-

gorithm. The first challenge is that Peng's algorithm has multiple recursive calls and the input to each recursive call depends on the output of previous recursive calls. These dependent recursive calls result in long dependent computation paths and thus the algorithm has a large (i.e. super-logarithmic) depth. The second challenge comes from how the approximate maximum flows are used in [RST14]. In particular, given an approximate maximum s-t flow, [RST14] crucially requires a decomposition of the flow into s-t flow paths.

Our paper's *main technical contribution* is to address these two challenges. To address the second challenge, we propose a novel efficient parallel flow decomposition routine which we discuss in more detail later in Section 5.1.2 of this overview. With this routine, we are already able to implement the algorithm of [RST14] with $m^{1+o(1)}$ work and $n^{o(1)}$ depth. However, getting the depth down to polylog(n) leaves us with the considerably more difficult task of addressing the first challenge. To this end, we devise a variant of the algorithm in [RST14] for constructing congestion approximators. While our new algorithm also computes a hierarchical decomposition of the graph, it allows us to apply our partitioning routine by running maximum flows on subgraphs obtained by *contracting* vertices, as opposed to recursing on vertex-induced subgraphs as done in [RST14]. This in particular allows us to *extract* congestion approximators for the subgraphs from a given congestion approximator of the entire graph, thus avoiding any additional recursive calls on these subgraphs. As a result, we only have *one* recursive call, whose goal is to compute a congestion approximator of the core graph C of the sparsifier, enabling us to obtain a depth of polylog(n).

We adopt a similar recursive idea to that of Peng [Pen16], with the key difference being our approach involves only one recursive call. Specifically, our construction consists of a *recursive step* to obtain a low-quality (though still polylog(n)-approximate) congestion approximator, and a *boosting step* to improve it to an $O(\log^9 n)$-congestion approximator, so as to avoid an accumulation of error over successive recursive calls. We summarize our new construction as follows:

1. Given input $G = (V, E, c)$, the goal is to output an $O(\log^9 n)$-congestion approximator tree $R = (V_R, E_R, c_R)$.
2. Compute a sparsifier H of G such that for any demand b, $\mathsf{opt}_H(b) \leqslant \mathsf{opt}_G(b) \leqslant \text{polylog}(n) \cdot \mathsf{opt}_H(b)$. In addition, H can be decomposed into a core graph C and a forest F such that

(i) C only has $|S|/\operatorname{polylog}(n)$ nodes and edges, and (ii) each connected component of F has exactly one vertex in C.

3. (Recursive step) **Only one recursive call:** Recursively compute an $O(\log^9 n)$-congestion approximator tree R_C for the core graph C.
4. Let $R_G = R_C \cup F$. Then R_G is a $\operatorname{polylog}(n)$-congestion approximator tree for G.
5. (Boosting step) Simulate our variant of the algorithm in [RST14] on G to find an $O(\log^9 n)$-congestion approximator tree R, where every time when $(1+\varepsilon)$-approximate minimum congestion flow computation is required on some subgraph $G(S)$ obtained from G by contracting vertices $V \setminus S$ into a single supernode, do the following:
   (a) Extract from R_G a $\operatorname{polylog}(n)$-congestion approximator tree R_S for $G(S)$.
   (b) Compute a $(1+\varepsilon)$-approximate minimum congestion flow required on $G(S)$ by plugging R_S into Sherman's algorithm.

Line 2 involves constructing ultrasparsifiers [ST14b, KMP14], and subsequently transforming them into j-trees [Mad10]. Though not particularly technically challenging, we are the first to give efficient constructions of these objects in the PRAM model. We refer the reader to Section 5.2 for their parallel implementations. Line 5, namely our variant of the algorithm in [RST14], is substantially more involved, as we must open the blackbox of [RST14] entirely. In the following section, we explain in more detail, our new construction of congestion approximators that is used in our boosting step.

5.1.1. Overview of New Congestion Approximator Construction

As mentioned above, the difficulty in parallelizing the construction of congestion approximators in [RST14] arises from the fact that we need to solve (approximate) maximum flows on vertex-induced subgraphs, which in turn requires congestion approximators for those subgraphs. However, recursing on these subgraphs necessarily blows up the depth of our algorithm to super-logarithmic. There are two possibilities of resolving this: (i) use a more aggressive size reduction than $\operatorname{polylog}(n)$ when computing sparsifiers, or (ii) reduce the number of dependent recursive calls. Here, (i) can be immediately ruled out, since a more aggressive size reduction implies a worse approximation

for the sparsifier, which in turn results in a larger iteration count (and hence, depth) of Sherman's algorithm to compute maximum flows. This leaves us with (ii) as the only option. Notice that having even just *two* dependent recursive calls would result in a depth of $2^{O\left(\frac{\log n}{\log \log n}\right)} = n^{o(1)}$, which makes (ii) even more imperative.

This raises the following question: can we reduce the number of dependent recursive calls by *reusing* congestion approximators? While it indeed seems plausible to combine the congestion approximators for the subgraphs into one for the entire graph, this does not suit the construction of [RST14] - in particular, the hierarchical decomposition tree there is computed in a *top-down manner*, and thus we do not know on what subgraphs we want to run maximum flows until we are done with the partitioning at higher levels. This suggests that we should instead consider *extracting* congestion approximators for the subgraphs *from* that of the entire graph. But unfortunately, this is in general impossible due to the information loss due to congestion approximation (which in turn is a form of sparsification that preserves both cuts and flows).

Our solution here is to open the blackbox of [RST14] entirely and propose a new framework for computing a hierarchical decomposition. Our new framework allows us to partition the graph by running (approximate) maximum flows on *contracted subgraphs*, each of which is obtained by contracting a subset of vertices into a single *supernode*, as opposed to vertex-induced subgraphs as in [RST14]. We show that for the contracted subgraphs we encounter in our construction, we are actually able to extract congestion approximators for them from a given congestion approximator of the entire graph by contracting its vertices[12], thereby avoiding *any* additional recursive calls. Crucially, the congestion approximators obtained for the contracted subgraphs have size proportional to the size of the corresponding subgraphs which prevents any substantial blow up in the total work of our algorithm.

We next highlight some additional ideas needed to make our new construction work. Since we run maximum flows on contracted subgraphs, the routing we get could use paths in the contracted

[12]This is a vast simplification of our extraction process. In particular, since we need our congestion approximator to be a tree in order to run Sherman's algorithm, we can only do certain "partial" contractions that preserves the tree structure, which significantly complicates both our algorithm and its analysis. We refer the reader to Section 5.4 for more detail.

subgraphs, which are not necessary valid in the entire graph. Therefore we need a "fixing" step in our routing - namely, whenever we obtain a routing in a contracted subgraph, we have to then fix it so that it becomes a valid routing (with low congestion) in the entire graph, while routing the exact same demands[13]. This fixing step is possible because our construction guarantees that the edges incident on a contracted supernode (which we call *boundary edges*) are always well-linked, in the sense that one could route in the entire graph a product multicommodity flow between them with low (polylog(n)) congestion. Thus, we can convert all the flow paths passing through a supernode in a contracted subgraph into flow paths in the entire graph using the well-linkedness of these edges while blowing up the congestion by at most polylog(n).

As stated thus far, it may seem that too many contracted subgraphs might end up using a same edge in the entire graph to route and thus overcongesting the edge by too much. We address this issue by fixing the routing level-by-level in our hierarchical decomposition in a bottom-up manner; at each level, we only partially fix the routing in a contracted subgraph, in the sense that it becomes valid (i.e. routes the exact same demand) in the slightly larger contracted subgraph obtained one level above. We aim to maintain the invariant that in each contracted subgraph, the edges incident on the contracted supernode (i.e. boundary edges) are never congested by a factor larger than polylog(n). This invariant guarantees that in the (partial) fixing step, the edges inside the contracted supernodes do not get congested by a factor larger than polylog(n) either: any routing that utilizes these edges has to go through the boundary edges in the first place, and the latter are guaranteed to have low congestion.

However, notice that naively performing this (partial) fixing operation level-by-level does not get us the desired invariant. This is because each time we use well-linkedness of the boundary edges to fix the routing, we get a multiplicative polylog(n) blowup in congestion, which accumulates across all $\Theta(\log n)$ levels to a very large value. To address this issue, we use a similar trick as in [RST14] where, when solving maximum flows in each contracted subgraph, we lower the weights of the boundary edges by a large enough polylog(n) factor. Therefore, the actual congestion we get on these edges is in fact polylog(n) times smaller than what we get in our maximum flow routing, canceling out

[13]In a contracted subgraph, we only ever route demands that are supported entirely on the "uncontracted" vertices. Thus there is no ambiguity in saying "same demands" in contracted subgraphs vs. in the entire graph.

the multiplicative blowup caused by the fixing step. Moreover, a polylog(n)-congestion approximator for the original contracted subgraph remains a polylog(n)-congestion approximator for the reweighted contracted subgraph, albeit with a worse polylog(n) approximation factor. Therefore we can still use Sherman's algorithm to solve maximum flows on the reweighted subgraph in polylog(n) depth. Leveraging all these ideas, we get a parallel algorithm for computing an $O(\log^9 n)$-congestion approximator.

5.1.2. Overview of Our Parallel Flow Decomposition Routine

We now summarize the ideas behind our parallel flow decomposition routine that addresses our second challenge. Here we are given an s-t flow represented by a flow network, with each edge carrying some non-negative amount of flow, and our goal is to decompose the flow into s-t flow paths. We will decompose the flow in an iterative manner where in each iteration, we *shortcut* flow paths of length two in the flow network by replacing them with a single edge having the same flow value. We then repeat until (almost) all remaining edges directly go from s to t (i.e. all s-t paths have length 1).

In order to achieve small depth, in every iteration, we need to find a large (in capacity) collection of edge-disjoint length two flow paths so that they can be shortcut in parallel without interfering with each other. We show that such a collection exists, and moreover, can be found efficiently by formulating the problem as b-matching instances. Specifically, for every (directed) edge $e = (u, v)$ that does not directly connect s and t, we independently and randomly assign e either as an "outgoing" edge to its head vertex u, or as an "incoming" edge to its tail vertex v. Now taking a local view at any vertex w in the flow network, the task of shortcutting reduces to a b-matching problem in a complete bipartite graph, where the two sides are vertices corresponding to incoming edges into w and outgoing edges from w, respectively, with demands equal to their flow values. This way, we never have to worry about an edge being matched more than once as it gets assigned to exactly one of its two endpoints.

We show that in expectation over the random incoming/outgoing assignments of the edges, the total weight of the maximum b-matchings across all vertices must be at least a constant fraction of the total remaining flow that does not directly go from s to t. Moreover, algorithmically, in each

b-matching instance, we can utilize a natural greedy strategy to gather at least a constant fraction of the maximum matched value, which can further be implemented in parallel across all vertices. This guarantees that in each iteration, the total ℓ_1-norm of the remaining flow shrinks by a constant factor, and thus after $O(\log n)$ iterations we have decomposed a $(1-1/\text{poly}(n))$-fraction of the total flow into s-t paths, which suffices for our purpose since we only want approximate maximum flows. At the end, we output the entire shortcutting history represented by a DAG data structure from which we can recover our desired information.

5.1.3. Algorithms

Lastly, we present all the key algorithms outlines in our overview. Our main algorithm is a recursive PRAM algorithm for computing a $O(\log^9 n)$-congestion approximator in $O(\text{polylog}\, n)$ depth and $\tilde{O}(m)$ total work. We present our algorithms for the case where the ratio between the largest and smallest capacities in the graph is bounded by $\text{poly}(n)$, but by the reduction shown in Section C.5, this is without loss of generality.

Algorithm 5.1.1 `congestion-approx`(G)

Input: Graph $G = (V, E, c)$ with $n = |V|, m = |E|$
Output: A $O(\log^9 n)$-congestion approximator for G.
Procedure:

1: $G_s \leftarrow$ `ultrasparsifier`(G, κ) with $\kappa = 10 \log^4 n$. ▷ Section 5.2.1.1
2: $(\mathcal{C}, \mathcal{E}) \leftarrow j$-`tree`$(G_s)$ with $\mathcal{C}$ the core and $\mathcal{E}$ the envelope of the j-tree. ▷ Section 5.2.1.2
3: // This is the only recursive call to `congestion-approx`
4: $R_{\mathcal{C}} \leftarrow$ `congestion-approx`$(\mathcal{C})$.
5: // $R_{\mathcal{C}} \cup \mathcal{E}$ is an $O(\log^{13} n)$-congestion approximator for G
6: $R' \leftarrow$ `tree-hierarchical-decomp`$(R_{\mathcal{C}} \cup \mathcal{E})$ ▷ Section 5.2.2
7: // From the call to `tree-hierarchical-decomp`, R' is a binary tree with $O(\log n)$ depth
8: $R \leftarrow$ `improve-CA`(G, R') ▷ Algorithm 5.1.2
9: **return** R

Our major technical contributions are in the new framework for computing congestion approximators, the implementation specifics of which are described in the `improve-CA` subroutine and the details of which are discussed in Section 5.3.

Theorem 5.1.1. *Given an undirected capacitated graph $G = (V, E, c)$ with n nodes and m edges, Algorithm 5.1.1 is a* $\text{polylog}\, n$*-depth, $O(m\ \text{polylog}\, n)$ total work PRAM algorithm which outputs a*

Algorithm 5.1.2 `improve-CA`(G, R)

Input: Graph $G = (V, E, c)$ and $O(\log^d n)$-congestion approximator R for G, with $9 < d \leqslant 30$.
Output: An $O(\log^9 n)$-congestion approximator for G.
Procedure:

1: $A_1, A_2 \leftarrow$ `partition-A`(G) ▷ Section 5.3.1
2: $B_1, B_2 \leftarrow$ `partition-B`(G, A_1, A_2) ▷ Section 5.3.2
3: // Claim 5.4.1 guarantees the edges leaving each Z_i are $O(1/\log^9 n)$-well-linked
4: Set $Z_1 = A_1 \cap B_1, Z_2 = A_2 \cap B_1, Z_3 = A_1 \cap B_2, Z_4 = A_2 \cap B_2$.
5: **for all** Z_i **do**
6: // R_i has size $\tilde{O}(|Z_i|)$ by Lemma 5.4.2
7: $R_i \leftarrow$ `ca-contraction`(R, Z_i) ▷ Section 5.4.1
8: // $G(Z_i)$ is a contracted subgraph (Definition 2.5.4), reweighted as in Section 5.3.3
9: $T_i \leftarrow$ `improve-CA`$(G(Z_i), R_i)$
10: Set R' as the tree formed by the T_i, using lines 11 and 12 of Algorithm 5.3.1.
11: **return** R'

hierarchical decomposition tree that, with high probability, is a $O(\log^9 n)$-congestion approximator for G.

Proof. The depth, work and correctness of all subroutines besides `improve-CA` follow from the proofs in their respective sections. Specifically, these sections show that R' passed to `improve-CA` is a $O(\text{polylog}\, n)$-congestion approximator for G which is a binary tree with depth $O(\log n)$, and that R' can be computed using $O(\text{polylog}\, n)$ depth and $\tilde{O}(m)$ work. Then, the correctness of `improve-CA` follows from Theorem 5.3.9, as does the depth and work of all but the `ca-contraction` calls. To bound the work from contracting the congestion approximators, by Lemma 5.3.3 we have that A_1, A_2, B_1, B_2 are both partitionings of V, and so $\bigcup_i Z_i = V$. Let $G(Z_i)$ be as defined in Definition 2.5.4, and properly modifying the capacities of some edges as in Section 5.3.3. As the Z_i's are a partitioning of nodes $V(G)$, each edge in G can be present in a most two contracted graphs $G(Z_i)$. Thus, $\sum_i |E(G(Z_i))| = O(m)$ as well. Constructing all contracted graphs $G(Z_i)$ can therefore be done in $O(m)$ work (and $O(1)$ depth), and so by Lemma 5.4.2, computing the contracted congestion approximators for all $G(Z_i)$ takes work $\tilde{O}(m)$ and depth $O(\log n)$. The total work and depth of the algorithm then follow from Theorem 5.3.9. □

Proof. (Theorem 7) Using the congestion approximator from Theorem 5.1.1, we can run the parallel version of Sherman's algorithm (Theorem 2.5.3) to get both a $(1 - \varepsilon)$-approximate maximum s-t

flow and a $(1+\varepsilon)$-approximate minimum s-t cut. □

5.1.4. Organization

We give details of our construction of congestion approximators in three subsequent sections. Firstly, Section 5.2 presents the construction of low-quality congestion approximators (outlined in Algorithm 5.1.1). Secondly, Section 5.3 presents the boosting step where the quality of the congestion approximator is improved (outlined in Algorithm 5.1.2). Finally, Section 5.4 presents details on how we extract congestion approximators for contracted subgraphs and utilize them within Sherman's framework [She13b]. We give details of our parallel flow decomposition algorithm in Section 5.5.

5.2. Recursive Computation of a Low-Quality Congestion Approximator

In this section, we detail the steps of Algorithm 5.1.1 up to the call to `improve-CA` (which is detailed in Algorithm 5.1.2 and Section 5.3). That is, in this section we detail how to recursively compute a α'-congestion approximator for G which is a $O(\log n)$ depth binary hierarchical decomposition tree of G where $\alpha' = O(\text{polylog}\, n)$. Section 5.3 then details how to improve the quality of the congestion approximator to $O(\log^9 n)$.

5.2.1. Reducing the Size of the Graph

5.2.1.1 Constructing an Ultrasparsifier.

To reduce the size, our first step is to construct an ultrasparsifier of the graph. An ultrasparsifier, informally speaking, consists of a spanning tree plus some additional edges that approximately preserves all graph cuts.

Definition 5.2.1 (Ultrasparsifier)**.** Given a graph $G = (V, E, c)$ with n nodes and m edges, and any parameter $\kappa \geqslant 1$, a κ-ultrasparsifier $G_s = (V, E_s, c_s)$ is the union of a spanning tree T of G and a collection of edges E' such that $|E'| = O(m \log^2 n/\kappa)$. Moreover, with high probability, the capacity of every cut of G is preserved to within a κ factor in G_s.

In Algorithm 5.1.1, we use $\kappa = 10 \log^4 n$ to ensure a $\text{polylog}\, n$ reduction in size compared to the original graph.

Lemma 5.2.2. *There exists a randomized PRAM algorithm that given an undirected, capacitated*

graph $G = (V, E, c)$, and any desired quality parameter $\kappa \geqslant 1$, constructs a κ-ultrasparsifier of G with $O(\log n)$ depth, and $O(m)$ total work.

Proof. The algorithm first computes a maximum spanning tree using the result of [PR02], which uses $O(\log n)$ depth and $O(m)$ work. Then, scale up the weights on the tree edges by a factor of κ, and sample each remaining edge with probability $\Theta(\log^2 n/\kappa)$.

Sampling can easily be done in parallel, and the number of sampled edges in expectation is also immediately as desired. So, it remains to show that the final constructed graph preserves all cuts to within a $O(\kappa)$ factor with high probability.

Let T be the constructed maximum spanning tree of G. Multiplying the weight of each edge of T by κ results in a worst-case cut distortion factor of κ. So, after scaling, we wish to select additional edges in order to approximate cuts (in the graph with newly scaled edges) to within an $O(1)$ factor. [FHHP19] shows that to approximate edges cuts to within a $(1+\varepsilon)$ factor, it is sufficient to sample each edge e with probability at least $Cc_e \log^2 n/(\rho_e \varepsilon^2)$, where ρ_e is the edge-connectivity between the endpoints of e, c_e is the capacity of edge e, and C is a suitably large constant. Here, the connectivity of an edge $e = (u, v)$ is defined as the capacity of the minimum cut separating u from v.

As our algorithm samples off-tree edges (that is, edges in G but not T) at rate $\Theta(\log^2 n/\kappa)$, to complete our analysis it suffices to show that the connectivity of any off-tree edge $e = (u, v)$ is at least $c_e\kappa$. Since e is off-tree, adding it to T creates a cycle. Moreover, T is the maximum weight spanning tree, so $c_e \leqslant c_{e'}$ for all e' on the cycle created. Any cut separating u from v must also involve some tree edge e' from this cycle, so the capacity of any cut (after scaling) is at least $c_{e'}\kappa$. Thus, the connectivity of e is also at least $c_{e'}\kappa \geqslant c_e\kappa$. □

5.2.1.2 Constructing a j-tree.

After constructing a $O(\text{polylog}(n))$-ultrasparsifier, we now have a graph G_s which is the union of a spanning tree T and collection of off-tree edges E'. We now transform the ultrasparsifer so that we only need to compute the congestion approximator for a smaller subgraph. Namely, we convert the ultrasparsifer into a j-tree. A j-tree, introduced in [Mad10], is a graph consisting of a forest

with j trees (called the *envelope*) connected together by an arbitrary graph (called the *core*) over j vertices, one from each tree.

Definition 5.2.3 (j-tree [Mad10])**.** A weighted graph $G = (V, E, c)$ is a j-tree if it is a union of a *core* graph H which is an arbitrary capacitated graph supported on at most j vertices of G, and a forest on V such that each connected component in the forest has exactly one vertex in H. We call this forest the *envelope* of the j-tree.

Lemma 5.8 of [Mad10] shows how to sequentially convert any ultrasparsifer to an $O(j)$-tree, where j is the number of nodes incident on off-tree edges in the ultrasparsifer. In this section, we show how to adapt this sequential process to the PRAM setting.

The algorithm of [Mad10] for converting an ultrasparsifier to a j-tree first recursively removes degree 1 vertices. Let F be the subgraph consisting of these deleted nodes and edges; F is thus a forest, and all removed edges come from the spanning tree T of the ultrasparsifier. Unfortunately, this process is inherently sequential, so we instead show that an alternative process, which can easily be parallelized and identifies this forest of removed nodes.

Lemma 5.2.4. *Let G_s be a $O(\text{polylog}(n))$-ultrasparsifier with spanning tree T and off-tree edges E'. Suppose T is rooted at an arbitrary node incident on some off-tree edge. Then, a node u is removed by recursive deletion of degree 1 vertices if and only if the subtree of T rooted at u contains no nodes incident on off-tree edges. Moreover, there is a PRAM algorithm with $O(\log n)$ depth and $O(m)$ work which removes all such nodes.*

Proof. Let r be the root of T, and let F be the set of nodes removed by recursively deleting degree 1 vertices (so F induces a forest). Define X as the set of nodes incident on some off-tree edge. If the subtree of u contains no nodes in X, then it immediately follows that u (and all of its descendants) must be in F because they induce a tree.

Now, suppose some node in the subtree of u is incident on an off-tree edge. Let p be the parent of u, let v be a child of u such that the subtree rooted at v also contains a node in X, and let H_u be the graph formed by removing u from G_s. As $r \in X$, both p and v are connected to X in H_u. If p, v are in the same connected component of H_u, then u is in a cycle, and so it will never be removed

by the recursive deletion procedure.

Otherwise, p, v are connected in H_u to distinct elements of X; p is connected to r and v is connected to some $x \in X$ such that $x \neq r$. As p and v are both neighbors of u, there exists in G_s a $r \to x$ path P such that $u \in P$. Consider the rounds of the recursive deletion procedure, where in each round the procedure removes all degree 1 vertices currently present in the graph. If before round i, no nodes of P have been deleted, then after round i, none of P has been removed as well: the interior points have 2 neighbors in the path, thus having degree at least 2, and the endpoints are never removed as they are incident on off-tree edges. The entire path is trivially present before the procedure begins, and so no nodes in P are ever removed. Thus, as $u \in P$, $u \notin F$, as desired.

The algorithm to find F is as follows: first, root T at an arbitrary node incident on some off-tree edge. Construct node weights by setting $w_u = 1$ if u is incident on some off-tree edge, and $w_u = 0$ otherwise. Then, run the subtree sum algorithm of Theorem C.1.4, and output all nodes whose subtree sum is 0. Identifying which nodes are adjacent to off-tree edges requires at most $O(1)$ depth and $O(m)$ work, and the subtree sum algorithm has depth $O(\log n)$ and total work $O(n)$, giving the desired runtime bounds. □

Let G' be the graph after recursively deleting degree 1 vertices, or, equivalently by Lemma 5.2.4, removing all nodes whose subtree contains no nodes incident on off-tree edges. The next step is to find paths in G' involving entirely degree 2 vertices except for the end points u, v which have degree at least 3, and "move" capacity from the lowest weight edge into an edge between the endpoints of the path. In Lemma 5.8, [Mad10] proves that after reconnecting the vertices removed by recusively deleting degree 1 vertices, this results in $(3j - 2)$-tree, where j is the number of nodes incident on off-tree edges in G_s.

Theorem 5.2.5 (Parallel Version of Lemma 5.8 of [Mad10])**.** *Algorithm 5.2.1 uses $O(\log n)$ depth and $\tilde{O}(m)$ total work to construct a $O(j)$-tree, where j is the number of nodes incident on an off-tree edge of G_s.*

Proof. For runtime, in Line 3 we may use the connectivity algorithm shown in [SV80], which has $O(\log n)$ depth and $\tilde{O}(m)$ work. Finding the additional neighbors can be identified in $O(1)$ depth

Algorithm 5.2.1 `transform-paths-and-cycles`(G')

Input: Graph G' with no degree 1 vertices, and j nodes incident on off-tree edges.
Output: A $O(j)$-tree G''.
Procedure:

Initialize $W \leftarrow \{u \mid \deg(u) = 2\}$, $\mathcal{P} = \emptyset, \mathcal{C} = \emptyset$, and $G'' = G'$.
Construct G_d by removing $V \setminus W$ (i.e. nodes of at least degree 3) from G'.
Run connectivity to identify connected components $C_1, \ldots C_q$ of G_d.
for each component C_i **do**
    Find $U_1 \subseteq C_i$, the set of degree 1 vertices in C_i.
    if $|U_1| = 0$ **then**
        Add C_i to $\mathcal{C}$.
    else
        Let $U_1 = \{u_1, u_2\}$, and let $v_1, v_2 \in V \setminus W$ be the additional neighbors in G' of u_1, u_2.
    if $v_1 = v_2$ **then**
        Add the cycle formed by C_i and v_1 to $\mathcal{C}$.
    else
        Add the path from $v_1 \to v_2$ through C_i to $\mathcal{P}$.
for each $S \in \mathcal{P} \cup \mathcal{C}$ **do**
    Find $e_{\min}$, the edge with minimum capacity in S.
    For all other $e \in S$, update capacity $w(e) = w(e) + w(e_{\min})$ in G''.
    If $S \in \mathcal{P}$ with endpoints (v_1, v_2) and $(v_1, v_2) \notin E(G'')$, create edge (v_1, v_2) with capacity $w(e_{\min})$.
    Delete $e_{\min}$ from G''.
return G''.

and $O(n)$ work, along with identifying whether they induce a path or a cycle, and a $O(\log n)$ depth algorithm can correctly update the capacities.

For correctness, all constructed paths have endpoints of degree 3. After removing all nodes of degree at least 3, the remaining nodes all have degree 2 in G', as G' has no degree 1 vertices by Lemma 5.2.4. So, every degree 2 vertex and every edge between degree 2 vertices is in exactly one component, and thus every edge is in some path or cycle and all paths and cycles are edge disjoint. The result then follows from the proof Lemma 5.8 in [Mad10]. □

Once we have a $O(j)$-tree J, to identify the core on which we wish to construct a congestion approximator, we first find a spanning tree of J. We then invoke the procedure of Lemma 5.2.4 with respect to this spanning tree to remove any nodes not present in the core. Then by Lemma 5.8 in [Mad10], the constructed j-tree preserves all cut values to within a polylog(n) factor.

Finally, we then recursively run Algorithm 5.1.1 on the core of the j-tree, and attach the trees in the envelope (where envelope is defined as in Definition 5.2.3).

5.2.2. Hierarchical Decomposition on Trees

After recursing on the core of the j-tree, we have a congestion approximator R' for G. However, the depth of R' may be very large, as there is no bound on the depth of trees which make up the envelope of the j-tree. Several parts of our algorithm, such as the procedure to reduce the size of the congestion approximator (Section 5.4.1), rely on R' having $O(\log n)$ depth. As such, before running the procedure to improve the quality of the congestion approximation of R', we emulate the hierarchical decomposition procedure of [RST14] *on the tree* R'. That is, we take as input the tree R', and output a hierarchical decomposition tree with depth $O(\log n)$ which is an $O(\log^4 n)$[14] congestion approximator for R'. Moreover, to bound the work of the contraction of congestion approximators in Section 5.4.1, we require the output congestion approximator to be a binary tree.

The goal of this section is to prove the following theorem:

Theorem 5.2.6 (Hierarchical Decomposition on Trees)**.** *Let T be a tree on n nodes. There is a*

[14] As this loss of $O(\log^4 n)$ suffices for our purposes, we do not attempt to improve it, and instead rely on the analysis of [RST14] in a near-black-box fashion.

$O(\log^3 n)$ depth, $\tilde{O}(n)$ work algorithm to compute an $O(\log^4 n)$-congestion approximator R for T which is a hierarchical decomposition with depth $O(\log n)$. Moreover, in the same depth and work, we can ensure that R is a binary tree.

The algorithm makes use of a *tree separator node*, which is any node q of a tree T such that removing q results in a forest of components of size at most $|T|/2$.

Definition 5.2.7 (Tree Separator Node)**.** A node q in a tree T is called a tree separator node of T if the forest induced by $T \setminus \{q\}$ consists of trees with at most $|T|/2$ nodes.

Lemma C.1.8 gives a $O(\log |T|)$ depth, $O(|T|)$ work PRAM algorithm to find a tree separator node for any tree T.

With this definition, we now describe the algorithm for computing a hierarchical decomposition on trees. Each level in the hierarchical decomposition algorithm of [RST14] consists of two partitioning steps. Given a set P of nodes, the first determines an $\Omega(1/\log^2 n)$-well-linked (see Definition 2.5.8) set of edges within the graph induced by P and for the second, it suffices to find an exact min-cut between these well-linked edges and the edges leaving P (see Lemma 3.9 of [RST14] for more details)[15].

Definition 5.2.8 (Partition A)**.** Let P be a subset of nodes of a graph $G = (V, E, c)$, and let $G'[P]$ be the subdivision graph (see Definition 2.5.6) of the subgraph of G induced by P. Then, Partition A is a partitioning of P into clusters $P_1, \ldots, P_w$ such that the set of inter-cluster edges in $G'[P]$, namely $\{(u, v) \mid u \in P_i, v \in P_j, i \neq j\}$, is $\Omega(1/\log^2 n)$-well-linked. Moreover, we have $|P_i| \leqslant (1/2)|V(G)|$ for all i.

For the definition of Partition B, we state it in terms of an exact min-cut, which is stronger than what [RST14] or Section 5.3 uses, since when the graph is a tree, we are able to compute exact min-cut efficiently in low depth (see Section C.4 for more details).

Definition 5.2.9 (Min-Cut Partition B)**.** Let P be a subset of nodes of a graph $G = (V, E, c)$, let B_P be set of edges with one endpoint in P and one endpoint in $V \setminus P$. Construct a graph H as follows. Start with $G'[P]$, and for each edge e in B_P, add a node x_e to H. Finally, for each

[15] We reuse the names "Partition A" and "Partition B" used in Section 5.3, to highlight the similar goals of the procedures `partition-A` and `partition-B`, however we are not invoking or directly implementing these procedures.

$e = (u, v)$ in B_P such that $v \in P$, add an edge (x_e, v) with capacity $c(e)/\log n$. Then, with W the set of nodes of H which corresponds to the set of subdivision nodes of the inter-cluster edges from Partition A, Partition B is a partitioning $(X, P \setminus X)$ such that $(X, P \setminus X)$ is a min-cut between B_P and W in the graph H.

The analysis of [RST14] and Section 5.3 shows that to get a polylog(n) depth, $\tilde{O}(n)$ work PRA-Malgorithm to compute a hierarchical decomposition for a tree, it suffices to implement Partition A and Partition B in polylog(k) depth and $\tilde{O}(k)$ work on a set of k nodes. We now give a pair of lemmas showing that we can indeed do this.

Lemma 5.2.10 (Computing Partition A on Trees)**.** *There is a $O(\log k)$ depth, $\tilde{O}(k)$ work PRAM algorithm to compute Partition A on a set of k nodes which induce a tree.*

Proof. Let T be the tree induced by the vertex set we wish to partition. Then, compute a tree separator node c of T, and set the partitions to be the connected components of $T \setminus \{c\}$ along with $\{c\}$. The depth and work are then $O(\log k)$ and $\tilde{O}(k)$, using the algorithms for finding a tree separator node and connected components.

The only edges cut are those incident on the center c. The set of edges incident on the center are 1-well-linked, since they share an endpoint, meeting the requirements for Partition A. □

Lemma 5.2.11 (Computing Partition B on Trees)**.** *Given a tree T, there is a $O(\log^2 k)$ depth, $\tilde{O}(k)$ work PRAM algorithm to compute Partition B on the subgraph of T induced by a set K of k nodes.*

Proof. First, note that the subgraph induced by K is a forest, as T is a tree. Then, with $P = K$, define B_P, W, and H as in Definition 5.2.9. Note that as the subgraph induced by K in T is a forest, H is also a forest. Add to H a super-source s connected to each node of B_P with capacity ∞ and a super-sink t connect to each node of W with capacity ∞. Then, the result follows from computing a exact s-t min-cut on H (using Lemma C.4.3)[16]. □

[16]Lemma C.4.3 is written for a tree with the addition of a source s and sink t, rather than a forest; as such, we first add a dummy node y that connects to one node of every component of the forest with a capacity 0 edge, which does not change the min-cut value and allows us to use Lemma C.4.3.

These lemmas allow us to compute hierarchical decompositions on trees.

Lemma 5.2.12. *Given a tree T on n nodes, there is an $O(\log^3 n)$ depth, $\tilde{O}(n)$ work PRAMalgorithm to compute an $O(\log^4 n)$-congestion approximator R for T which is a hierarchical decomposition with depth $O(\log n)$.*

Proof. We follow the algorithm of [RST14] to construct a congestion approximator for T. As each level of the constructed congestion approximator in [RST14] corresponds to a partitioning of the edges, computing Partition A and B for all sets at a given level can be done in $O(\log^2 n)$ depth and $\tilde{O}(n)$ work. The use of tree separator nodes leads to a constant factor size reduction from each Partition A step, and so there are at most $O(\log n)$ levels, leading to a $O(\log^3 n)$ depth, $\tilde{O}(n)$ work algorithm. Finally, the analysis of [RST14] shows that the constructed hierarchical decomposition is a $O(\log^4 n)$-congestion approximator for the original tree, and that the tree has $O(\log n)$ depth. □

It thus remains shows how to convert the congestion approximator R into a binary tree; we describe the procedure in Algorithm 5.2.2. In Algorithm 5.2.2, we use the term *first non-binary node* to refer to any node $u \in R$ such that none of the ancestors of u have more than two children, but u has at least three children. Since R has depth $O(\log n)$, a breadth-first traversal can find all first non-binary nodes in $O(\log n)$ depth and $O(n)$ work.

Finally, for any node u of a hierarchical decomposition tree R, we use the notation R_u to refer to the subtree of R rooted at u, and P_u the set represented by node u.

To prove the correctness of Algorithm 5.2.2, we first prove the following helper lemma.

Lemma 5.2.13. *Let $R' =$ `convert-binary`(R), and consider x a node of R'. Let y be the parent of x, and let z be the parent of y. Then, $|P'_x| \leqslant (3/4)|P'_z|$, where for any $w \in R'$, P'_w is the set represented by w in R'.*

Proof. Every node in R is present in R', as each call to `convert-binary` adds nodes but does not remove any. As such, $V(R) \subseteq V(R')$. We again use the notation P_w to denote the set represented by a node w in R, and $P'_w = P_w$ for all $w \in R$. Call any node in $R' \setminus R$ an intermediary node. First, consider a node x in R' whose parent y is not an intermediary node; that is, $y \in R$. If $x \in R$ as

Algorithm 5.2.2 `convert-binary`(R)

Input: Hierarchical decomposition tree $R = (V_R, E_R, c_R)$ which is a α-congestion approximator for some tree T.

Output: Binary tree R' which is a α-congestion approximator for T.

Procedure:

1: Compute the set $\mathcal{B}$ of all first non-binary nodes in R

2: // R_u is the subtree of R rooted at u

3: Let $\mathcal{F} = \bigcup_{u \in \mathcal{B}} R_u \setminus \mathcal{B}$ be the forest of descendants of all first non-binary nodes

4: Initialize $R' \leftarrow R \setminus \mathcal{F}$

5: **for** each first non-binary node $u \in \mathcal{B}$ **do**

6: // P_{v_i} is the set represented by v_i in R, and $q \geqslant 3$ as $u \in \mathcal{B}$

7: Let $v_1, \ldots, v_q$ be the children of u such that $|P_{v_1}| \geqslant |P_{v_2}| \geqslant \ldots |P_{v_q}|$

8: Set s such that s is the smallest index such that $|P_{v_1} \cup \ldots \cup P_{v_s}| \geqslant |P_u|/4$

9: Add to R' new nodes z_1, z_2 as children of u with edges of capacity ∞. Moreover, in R', z_1 represents $P_{v_1} \cup \ldots \cup P_{v_s}$ and z_2 represents $P_{v_{s+1}} \cup \ldots \cup P_{v_q}$

10: **for** $1 \leqslant i \leqslant s$ **do**

11: Add R_{v_i} to R' by adding edge (z_1, v_i) with capacity $c_R((u, v_i))$

12: **for** $s + 1 \leqslant i \leqslant q$ **do**

13: Add R_{v_i} to R' by adding edge (z_2, v_i) with capacity $c_R((u, v_i))$

14: If z_1 (or z_2) has exactly one child v, remove z_1 (or z_2) and connect v to u with an edge of capacity $c_R((u, v))$.

15: If R' is binary, return R'. Otherwise, return `convert-binary`(R').

well, then by Lemma 5.2.10, $|P_x| \leqslant (1/2)|P_y|$. So, suppose x is an intermediary node, and thus P'_x is a union of sets $P_{w_1}, \ldots, P_{w_q}$, where $w_1, \ldots, w_q$ are the children of y in R. As in Algorithm 5.2.2, set s to be the smallest index such that $|P_{w_1} \cup \ldots \cup P_{w_s}| \geqslant (1/4)|P_y|$, and let $S_1 = P_{w_1} \cup \ldots \cup P_{w_s}$ and $S_2 = P_{w_{s+1}} \cup \ldots \cup P_{w_q}$. From the fact that $|P_{w_i}| \leqslant (1/2)|P_y|$ (which is again from Lemma 5.2.10) and the setting of s, we have that $(1/4)|P_y| \leqslant |S_1| \leqslant (3/4)|P_y|$. Since $S_1 \cup S_2 = P_y$, the same inequality holds for $|S_2|$. Thus, as either $P'_x = S_1$ or $P'_x = S_2$, we have that $|P'_x| \leqslant (3/4)|P'_y|$. Since R' is a hierarchical decomposition, if z is the parent of y, $P'_y \subseteq P'_z$, and thus all nodes which are the child of a non-intermediary node satisfy the lemma.

Now, consider a node $x \in R'$ whose parent y is an intermediary node. Let z be the parent of y. As R' is a hierarchical decomposition by construction, there exists a set $S' \subseteq V(R)$ such that $P'_z = \bigcup_{w \in S'} P_w$ and a set $S \subset S'$ such that $P'_y = \bigcup_{w \in S} P_w$. If for all $w \in S$, $|P_w| \leqslant (1/2)|P'_z|$, then by the same argument used in the case where y was non-intermediary, $|P'_x| \leqslant |P'_y| \leqslant (3/4)|P'_z|$. Suppose, for contradiction, there exists a $w \in S$ such that $|P_w| > (1/2)|P_z|$. Since Algorithm 5.2.2 sorts the nodes in Line 7 in decreasing order of size of the sets they represent, if S contains an element w such that $|P_w| > (1/2)|P_z|$, w must be the only element of S. As such, since $S = \{w\}$, y is not an intermediary node, which is a contradiction. □

Lemma 5.2.14. *Given an α-congestion approximator R from Lemma 5.2.12 for a tree T with n nodes, Algorithm 5.2.2 is a $O(\log^2 n)$ depth, $\tilde{O}(n)$ work PRAM algorithm that constructs a binary tree R' such that R' is a α-congestion approximator for T. Moreover, R' has depth $O(\log n)$.*

Proof. Let $R' = \texttt{convert-binary}(R)$ be the output of Algorithm 5.2.2 on R. Lemma 5.2.13 immediately implies that R' has depth $O(\log n)$, and Line 15 guarantees that it is binary, so it remains to show that R' is a α-congestion approximator for T and that Algorithm 5.2.2 has $O(\log^2 n)$ depth and $\tilde{O}(n)$ total work. Note that for any edge (u, v) in R, there is a path from u to v in R' with the same minimum capacity as (u, v) in R; namely, all edges on the path from u to v in R' have capacity either ∞ or $c_R((u, v))$. Let P_{uv} be the path for (u, v) in R', and let P_{xy} be the path for some other edge $(x, y) \in E(R)$ in R'. By the setting of edge capacities in Algorithm 5.2.2, all edges along both P_{uv} and P_{xy}, if there are any, must have capacity ∞. As such, any routing in R' can be

converted to a routing in R with the same congestion, and vice-versa, so R' is also a α-congestion approximator for T.

Let $R_0 = R, R_1, \ldots, R_L$ be the sequence of trees such that for all $i < L$, `convert-binary`(R_i) makes a recursive call `convert-binary`(R_{i+1}) in Line 15, and the call `convert-binary`(R_L) returns R' without further recursion. Each call to `convert-binary` only increases the number of nodes and depth of the tree: no nodes are removed, and each addition of a node z_i in Line 9 can only increase the depth. So, it follows that for any $i \in [L]$, the depth of R_i is no more than the depth of R'; by Lemma 5.2.13, we have that the depth of R_i is then $O(\log n)$. Moreover, by construction, each R_i is a hierarchical decomposition of T, so each level j of R_i represents a partitioning of $V(T)$. Thus, as there are $O(\log n)$ levels, each node in T can appear in the sets of at most $O(\log n)$ nodes of R_i, each at a different level of R_i. Since there are n nodes in T, it follows that there are $O(n \log n)$ nodes in R_i.

Each call `convert-binary`(R_i) before the recursive call in Line 15 can be implemented in $O(\log |R_i|)$ depth and $\tilde{O}(|R_i|)$ work. Since $|R_i| = O(n \log n)$, each call can be implemented in $O(\log n)$ depth and $\tilde{O}(n)$ work. To bound the total depth and work, it thus suffices to bound the total number of recursive calls to `convert-binary` made during a call of `convert-binary`(R). Define $B(R_i)$ to be the lowest depth (i.e. distance from the root) of any first non-binary node in R_i. In each call `convert-binary`(R_i), after the `for` loop of Line 4, all first non-binary nodes have at most two children, and are thus no longer first non-binary nodes. In addition, the `for` loop can only reduce the number of children any node has. As such, it follows that $B(R_{i+1}) \geqslant B(R_i) + 1$ for all $i < L$. Then, as R_L has depth $O(\log n)$, we must have $B(R_i) = O(\log n)$ for all i, and thus $L = O(\log n)$ as well. This results in a complete depth of $O(\log^2 n)$ and total work of $\tilde{O}(n)$. □

The proof of Theorem 5.2.6 then follows from Lemma 5.2.12 and Lemma 5.2.14.

5.3. New Framework for Congestion Approximator Computation

The goal of this section and the next (Section 5.4) is to give a full presentation and analysis of our Algorithm 5.1.2, whose goal is to boost the approximation quality of a given congestion approximator. Specifically, given any congestion approximator of a graph G of arbitrary polylog(n)

distortion, we will compute an $O(\log^9 n)$-congestion approximator of G in $O(m \operatorname{polylog}(n))$ work and $O(\operatorname{polylog}(n))$ depth.

Our presentation of Algorithm 5.1.2 will be given as two parts. First in this section, we show, assuming being able to solve approximate maximum flows on contracted subgraphs, how we can compute a congestion approximator of $O(\log^9 n)$ distortion. Then in the next section (Section 5.4), we show how we can directly extract congestion approximators for contracted subgraphs from a given congestion approximator of the entire graph, allowing us to then run Sherman's algorithm [She13b] to compute $(1-\varepsilon)$-approximate maximum flows on these contracted subgraphs.

We now present in this section a variant of the framework in [RST14] that allows us to *boost* the approximation of a given congestion approximator in a manner that is both work-efficient and parallelizable. The key novelty in our new framework is to avoid running (approximate) maximum flows on subgraphs of G, which are obtained by removing vertices/edges from G. This is because running maximum flows on such subgraphs requires computing congestion approximators for them using additional recursions, which would have blown up the depth of our algorithm to at least $n^{o(1)}$. Rather, we run approximate maximum flows on *contracted subgraphs* of G, that are obtained from G by contracting subsets of vertices into single nodes. As we will show in Section 5.4, we can extract congestion approximators for contracted subgraphs *directly* from a given congestion approximator of the entire graph, *without* having to recurse on the subgraphs.

Throughout this section, we assume that we have a PRAM algorithm $\mathcal{A}$, whose implementation we describe in the next section (Section 5.4), that can compute a $(1-\varepsilon)$-approximate maximum flow on any given contracted subgraph of G with work near-linear in the number of edges of the subgraph and depth $O(\operatorname{polylog}(n))$. We separate the implementation of $\mathcal{A}$ from the presentation of the new framework here in an effort to make the latter cleaner and more intelligible.

Remark 5.3.1. *Note that the specific implementation of algorithm $\mathcal{A}$ will depend on the execution of our framework in this section, and in particular the execution of our framework and that of $\mathcal{A}$ should alternate with each other. This is because the implementation of $\mathcal{A}$ involves extracting congestion approximators for the contracted subgraphs from the given congestion approximator of the entire graph. However, only after we have run $\mathcal{A}$ on a current cluster S do we know the sub-*

clusters on which we want to run $\mathcal{A}$ subsequently. The composition of the two algorithms is given in Algorithm 5.1.2.

We state the performance of our assumed PRAM algorithm $\mathcal{A}$ below, and describe how to implement it in Section 5.4. Note that in our algorithm, we always set ζ to be at least $1/\operatorname{polylog}(n)$ so $\mathcal{A}$ has near-linear total work and $\operatorname{polylog}(n)$ depth.

Proposition 5.3.2 (Performance of $\mathcal{A}$)**.** *Let $G'(\mathcal{P} = (S, \omega))$ be the subdivison of a contracted subgraph of G with reweighting function ω such that the range of ω is within $[\zeta, 1]$ for some $\zeta \in (0, 1]$. Then given an arbitrary graph $G'_{st}(\mathcal{P})$ obtained from $G'(\mathcal{P})$ with maximum s-t flow value F^*, $\mathcal{A}$ computes with high probability*

1. *A feasible s-t flow of value at least $(1-\varepsilon)F^*$ in $G'_{st}(\mathcal{P})$.*
2. *An s-t cut $(T, \bar{T})$ in $G'_{st}(\mathcal{P})$ with capacity at most $(1+\varepsilon)F^*$.*

$\mathcal{A}$ has total work $O(|E(G'_{st}(\mathcal{P}))|\zeta^{-2}\varepsilon^{-3}\operatorname{polylog}(n))$ and depth $O(\zeta^{-2}\varepsilon^{-3}\operatorname{polylog}(n))$.

Plan for the Rest of the Section. We will obtain a hierarchical decomposition tree of the graph that serves as our congestion approximator. In each step of the decomposition, we use the maximum flow algorithm $\mathcal{A}$ to perform two partitioning steps on a current cluster S, and use the resulting partitions to obtain a two-level decomposition tree. Recursively applying this decomposition step to each sub-cluster obtained ends up giving us a hierarchical decomposition tree of the graph. As will be guaranteed by our partitioning steps, in each decomposition step we reduce the size of the cluster by a constant factor. As a result, we get a tree of $O(\log n)$ depth.

At a high level, the goals of the two partitioning steps are similar to those of [RST14]. Specifically, the first partitioning step is to find a subset of edges that (i) are well-linked in the sense that we can route product demands between them with low congestion, and (ii) separate the current cluster S into sub-clusters whose sizes are a constant factor smaller. The second partitioning step is then to find a bottleneck cut that separates the inter-cluster edges found in step one and the *boundary edges* that go from S to $V(G) \setminus S$. The difference between our partitioning steps and those of [RST14] is that we obtain these partitions in contracted subgraphs rather than vertex-induced subgraphs, which avoids additional recursions on these subgraphs when solving maximum flows on them, but

on the other hand requires extra care so as to prevent the edges from getting over-congested, since after all an edge could implicitly appear in polynomially many contracted subgraphs as one inside the contracted vertices.

In the rest of this section, we will first present the two partitioning steps needed, and then show how to use them to obtain a hierarchical decomposition tree. In the following, we will interchangeably use the terms *partitioning* and *clustering*, and the terms *partitions* and *clusters*. For a given edge subset F, the partition *induced by* F is the partition of the vertices correponding to the connected components of the graph after the removal of F.

5.3.1. The First Partitioning Step

Given a contracted subgraph $G(\mathcal{P})$ with at least two uncontracted vertices, our first step is to partition the vertices $V(G(\mathcal{P}))$ into sets Z_1, Z_2 such that the edges between Z_1, Z_2 are well-linked in $G(\mathcal{P})$, and Z_1, Z_2 are size balanced. To this end, we first use the parallel implementation of the cut-matching game in [RST14] to get a balanced partition with *almost* all of the inter-cluster edges well-linked. To simplify our presentation, we say a partition $Z_1, \ldots, Z_z$ of a set Z is γ-balanced for some $\gamma \in (0, 1]$ if $|Z_i| \leqslant \gamma|Z|$ for all i.

By plugging $\mathcal{A}$ into the cut matching game of [RST14] (their Lemma 3.1), whose parallel version is described in Appendix C.2, we have the following lemma.

Lemma 5.3.3. *There exists a PRAMalgorithm* `partition-A1` *that given $G(\mathcal{P})$ for $\mathcal{P} = (S, \omega)$ with $X = V(G) \setminus S$ and the range of ω within $[\zeta, 1]$, and a set of edges $F \subseteq E$ that induces a $3/4$-balanced partition of $V(G(\mathcal{P}))$, with high probability computes a set of new edges F_{new} such that F_{new} also induces a $3/4$-balanced partition of $V(G(\mathcal{P}))$, and*

1. *either $c^{G(\mathcal{P})}(F_{\text{new}}) \leqslant \frac{7}{8} c^{G(\mathcal{P})}(F)$;*
2. *or $F_{\text{new}} = A \cup R$ with A, R disjoint, such that $c^{G(\mathcal{P})}(A) \leqslant c^{G(\mathcal{P})}(F)$, $c^{G(\mathcal{P})}(R) \leqslant \frac{2}{\log n} \cdot c^{G(\mathcal{P})}(A)$, and edges in A are β-well-linked in $G(\mathcal{P})$ for $\beta = \Omega(1/\log^2 n)$.*

The algorithm `partition-A1` *has $O(|E(G(\mathcal{P}))|\zeta^{-2} \operatorname{polylog}(n))$ total work and $O(\zeta^{-2} \operatorname{polylog}(n))$ depth.*

We also need the following lemma, whose proof is deferred to Section 5.3.2.

Lemma 5.3.4. *There is a PRAMalgorithm* `partition-A2` *that given $G(\mathcal{P})$ for $\mathcal{P} = (S, \omega)$ with $X = V(G) \setminus S$ and the range of ω within $[\zeta, 1]$, a set of edges $F = A \cup B$ inducing a 3/4-balanced partition of $V(G(\mathcal{P}))$ where A, B are disjoint with $c^{G(\mathcal{P})}(B) \leqslant 2c^{G(\mathcal{P})}(A)/\log n$,*

1. *either finds an edge set F_{new} with $c^{G(\mathcal{P})}(F_{\text{new}}) \leqslant \frac{3}{4}c^{G(\mathcal{P})}(F)$ that induces a 3/4-balanced partition;*
2. *or finds an edge set C disjoint from A such that $F_{\text{new}} := A \cup C$ induces a 3/4-balanced partition, and there exists a multicommodity flow in $G'(\mathcal{P})$ with congestion $O(\log n)$ from X_C to X_A such that (i) each $x_e \in X_C$ sends $c_e^{G(\mathcal{P})}$ units of flow, and (ii) each $x_f \in X_A$ receives $O(\log n) \cdot c_f^{G(\mathcal{P})}$ units of flow.*

The algorithm `partition-A2` *has $O(|E(G(\mathcal{P}))|\zeta^{-2}\operatorname{polylog}(n))$ total work and $O(\zeta^{-2}\operatorname{polylog}(n))$ depth.*

We now prove the main lemma about our first partitioning step.

Lemma 5.3.5. *There is a PRAMalgorithm* `partition-A` *that given $G(\mathcal{P})$ for $\mathcal{P} = (S, \omega)$ with $X = V(G) \setminus S$ and the range of ω within $[\zeta, 1]$, with high probability partitions the uncontracted vertices into Z_1, Z_2 such that*

1. *$Z_1 \cup Z_2 = V(G(\mathcal{P})) \setminus \{u_X\}$.*
2. *$|Z_i| \leqslant \frac{7}{8}|V(G(\mathcal{P}))|$ for each $i = 1, 2$.*
3. *The set of inter-cluster edges $F := \{(u, v) | u \in Z_1, v \in Z_2\}$ is β-well-linked in $G(\mathcal{P})$ for $\beta = \Omega(1/\log^3 n)$.*

The algorithm `partition-A` *has $O(|E(G(\mathcal{P}))|\zeta^{-2}\operatorname{polylog}(n))$ total work and $O(\zeta^{-2}\operatorname{polylog}(n))$ depth.*

Proof. By composing Lemmas 5.3.3 and 5.3.4 the same way as [RST14] compose their Lemmas 3.1 and 3.2, we get in desired total work and depth a partition $Z'_1, \ldots, Z'_z$ of $V(G(P))$ such that

1. $Z'_1 \cup \ldots \cup Z'_z = V(G(\mathcal{P}))$.
2. $|Z'_i| \leqslant \frac{3}{4}|V(G(\mathcal{P}))|$ for each $i \in [z]$.
3. The set of inter-cluster edges $F' = \left\{(u, v) | u \in Z'_i, v \in Z'_j, i \neq j\right\}$ is β'-well-linked in $G(\mathcal{P})$ for

$\beta' = \Omega(1/\log^3 n)$.

We can then obtain Z_1, Z_2 by the following process. First, we remove the supernode u_X from $Z'_1, \ldots, Z'_z$ to obtain a partition $Z''_1, \ldots, Z''_z$ of the uncontracted vertices. Now the inter-cluster edges between $Z''_1, \ldots, Z''_z$ are still $\Omega(1/\log^3 n)$-well-linked in $G(\mathcal{P})$ since they are a subset of the inter-cluster edges between $Z'_1, \ldots, Z'_z$. Then to get a bi-partition, we merge the subsets $Z''_1, \ldots, Z''_z$ as follows. Let i be the largest integer s.t. $|Z''_1 \cup Z''_2 \cup \ldots \cup Z''_i| \leqslant \frac{7}{8}|V(G(\mathcal{P})|$. Then we merge the $Z''_1, \ldots Z''_i$ into one partition Z_1 and the remaining Z''_i's into another partition Z_2 to obtain our bi-partition Z_1, Z_2. Then because $Z''_1, \ldots, Z''_z$ is a partition of the uncontracted vertices, so is Z_1, Z_2. Since each Z''_i has size at most $\frac{3}{4}|V(G(\mathcal{P}))|$, by the definition of i, Z_1 has size between $[\frac{1}{8}|V(G(\mathcal{P}))|, \frac{7}{8}|V(G(\mathcal{P}))|]$. Therefore, Z_2 has size at most $\frac{7}{8}|V(G(\mathcal{P}))|$ since Z_1, Z_2 is a partition of the uncontracted vertices in $V(G(\mathcal{P}))$. Finally, inter-cluster edges (call them F) from Z_1 to Z_2 are still $\Omega(1/\log^3 n)$-well-linked in $G(\mathcal{P})$ since they are a subset of the inter-cluster edges between $Z''_1, \ldots, Z''_z$.

Note that the partitioning into Z_1, Z_2 can be done in parallel in $O(\log n)$ depth by computing a prefix sum and a binary search. The total work and depth then follows from the performance of the first partitioning step of [RST14]. □

5.3.2. The Second Partitioning Step

Let B denote the *boundary edges* in $G(\mathcal{P})$, namely those that go from the contracted vertex u_X to the uncontracted vertices. In other words, $B := E(u_X, V(G) \setminus X)$.

Our second step is to find a cut in $G(\mathcal{P})$ separating the boundary edges B from the inter-cluster edges F that we identified in the first partitioning step. Here, we want the property that there is a low-congestion routing from the cut edges we find to the boundary edges B, as well as from the cut edges to the inter-cluster edges F, such that each cut edge sends out flow equal to its capacity. Specifically, we prove the following lemma. Recall that for a graph H and an edge $e \in H$, we write c_e^H to denote the capacity of edge e in H.

Lemma 5.3.6. *There is a PRAMalgorithm* `partition-B` *that given $G(\mathcal{P})$ for $\mathcal{P} = (S, \omega)$ with $X = V(G) \setminus S$ and the range of ω being within $[\zeta, 1]$ with $\zeta \in (0, 1]$, two disjoint edge subsets*

$B, F \subset E(G(\mathcal{P}))$, and a parameter $\psi \in (0, 1]$, with high probability returns a subset of edges Y (potentially intersecting both B and F) in $G(\mathcal{P})$ such that

1. *X_Y separates X_B from X_F in $G'(\mathcal{P})$, that is, in $G'(\mathcal{P})$ every path between a vertex in X_B and a vertex in X_F must contain a vertex in X_Y. Moreover, in $G(\mathcal{P})$, the total capacity of Y is at most twice the capacity of B and at most twice the capacity of F.*
2. *There is a multicommodity flow in $G'(\mathcal{P})$ from X_Y to X_B such that*
   (a) *The congestion on edges incident on u_X in $G'(\mathcal{P})$ is $O(\psi \log n)$, while the congestion on other edges in $G'(\mathcal{P})$ is $O(\log n)$.*
   (b) *Each $x_y \in X_Y$ sends $c_y^{G(\mathcal{P})}$ units of flow while each $x_b \in X_B$ receives at most $O(\log n) \cdot c_b^{G(\mathcal{P})}$ units of flow.*
3. *Similarly, there is a multicommodity flow in $G'(\mathcal{P})$ from X_Y to X_F such that*
   (a) *The congestion on edges incident on u_X in $G'(\mathcal{P})$ is $O(\psi \log n)$, while the congestion on other edges in $G'(\mathcal{P})$ is $O(\log n)$.*
   (b) *Each $x_y \in X_Y$ sends $c_y^{G(\mathcal{P})}$ units of flow, while each $x_f \in X_F$ receives at most $O(\log n) \cdot c_f^{G(\mathcal{P})}$ units of flow.*

`partition-B` has $O(|E(G(\mathcal{P}))|\zeta^{-2}\psi^{-2}\operatorname{polylog}(n))$ total work and $O(\zeta^{-2}\psi^{-2}\operatorname{polylog}(n))$ depth.

Proof. We will do an iterative refinement process to find the desired set of cut edges Y. Initially, we start with Y being the smaller (in capacity) of B and F, which has the desired property that X_Y separates X_B from X_F in $G'(\mathcal{P})$. We will maintain this property, while "refine" the set Y. In particular, we will classify the edges in Y into *good* edges Y_{good} and *bad* edges Y_{bad}, and try to reduce the total capacity of the bad edges Y_{bad}. Initially, $Y_{\text{good}} = \emptyset$ and $Y_{\text{bad}} = Y$.

In each iteration, we do the following refinement step. Define a reweighting function $\tilde{\omega}$ on the split edges (in $G'(\mathcal{P})$) of the boundary edges of $G(\mathcal{P})$ such that

$$\tilde{\omega}(f) = \begin{cases} \psi & f \text{ incident on } u_X \\ 1 & \text{otherwise.} \end{cases}$$

and look at the graph $G'(\tilde{\mathcal{P}} = (\mathcal{P}, \tilde{\omega}))$. Let $\varepsilon = \log^{-1} n$.

Let $T_1 := B$ and $T_2 := F$. For $j \in \{1, 2\}$, we consider the graph $G'_{s_j t_j}(\tilde{\mathcal{P}})$ where we connect s_j to each $x_y \in X_{Y_{\text{bad}}}$ with capacity $c_y^{G(\mathcal{P})}$ and connect each $x_e \in X_{T_j}$ to t_j with capacity $c_e^{G(\mathcal{P})}$. We run $\mathcal{A}$ (as defined in Proposition 5.3.2) on $G'_{s_j t_j}(\tilde{\mathcal{P}})$ to find a $(1-\varepsilon)$-approximate maximum s_j-t_j flow f'_j and a $(1+\varepsilon)$-approximate s_j-t_j minimum cut with edges Y'_j.

We say an edge $y \in Y_{\text{bad}}$ *has become good*, if for both $j = 1, 2$, the edge (s_j, x_y) in $G'_{s_j t_j}(\tilde{\mathcal{P}})$ carries (in the direction from s_j to x_y) at least $c_y^{G(\mathcal{P})}/4$ units of flow in f'_j.

We then do a case analysis as follows:

Case 1 If for both $j = 1, 2$, we have

$$c^{G'_{s_j t_j}(\tilde{\mathcal{P}})}(Y'_j) \geqslant \frac{9}{10} c^{G(\mathcal{P})}(Y_{\text{bad}}),$$

then move the edges in Y_{bad} that have become good to Y_{good}.

Case 2 Else, let $j^* \in \{1, 2\}$ be such that

$$c^{G'_{s_{j^*} t_{j^*}}(\tilde{\mathcal{P}})}(Y'_{j^*}) < \frac{9}{10} c^{G(\mathcal{P})}(Y_{\text{bad}}).$$

Then let $Y_{\text{bad}} \leftarrow Y_{j^*}$, where Y_{j^*} is constructed as follows:

(a) Let Y''_{j^*} be Y'_{j^*} with edges incident on u_X removed.

(b) Include any edge $e \in E(G(\mathcal{P}))$ in Y_{j^*} if x_e is incident on at least one edge in Y''_{j^*}.

Claim 5.3.7. *In **Case 1**, the edges $y \in Y_{\text{bad}}$'s that have become good contribute at least $1/5$ of the total capacity of Y_{bad}.*

Proof. For $j \in \{1, 2\}$, consider the flow f'_j in $G'_{s_j t_j}(\tilde{\mathcal{P}})$. The flow value is at least $\frac{4}{5} c^{G(\mathcal{P})}(Y_{\text{bad}})$ given the conditions of **Case 1** and that f'_j is a $(1-\varepsilon)$-approximate maximum flow. Therefore at least $3/5$ (in capacity) of $y \in Y_{\text{bad}}$'s (call them $Y^{(j)}_{\text{good}}$) satisfies that the edge (s_j, x_y) in $G'_{s_j t_j}(\tilde{\mathcal{P}})$ carries (in the direction from s_j to x_y) at least $c_y^{G(\tilde{\mathcal{P}})}/4$ units of flow in f'_j, since otherwise the total flow

value would be

$$\left(\frac{3}{5}+\frac{2}{5}\cdot\frac{1}{4}\right)c^{G(\mathcal{P})}(Y_{\text{bad}})<\frac{4}{5}c^{G(\mathcal{P})}(Y_{\text{bad}}),$$

a contradiction. The claim then follows by taking the intersection of $Y^{(1)}_{\text{good}}$ and $Y^{(2)}_{\text{good}}$. □

Claim 5.3.8. *In **Case 2**, we have*

$$c^{G(\mathcal{P})}(Y_{j^*})\leqslant c^{G'_{s_{j^*}t_{j^*}}(\tilde{\mathcal{P}})}(Y'_{j^*}).$$

Moreover, $X_{Y_{j^*}\cup Y_{\text{good}}}$ separates X_B from X_F.

Proof. For the capacity condition, notice that we can simply charge the capacity of every edge in $e\in Y_{j^*}$ to the capacity of an edge in Y''_{j^*} that is incident on x_e. Let Y'_{good} denote the set of all split edges of edges in Y_{good} in $G'(\mathcal{P})$. Define edge set Z_{j^*} by including edges $e\in E(G(\mathcal{P}))$ for which x_e is incident on at least on edge in Y'_{j^*}. Since Y'_{j^*} is an s_j-t_j cut in $G'_{s_jt_j}(\tilde{\mathcal{P}})$, $X_{Z_{j^*}}$ separates $X_{Y_{\text{bad}}}$ from $X_{T_{j^*}}$. Since $X_{Y_{\text{bad}}}\cup X_{Y_{\text{good}}}$ separates X_B and X_F, $X_{Z_{j^*}}\cup X_{Y_{\text{good}}}$ also separates X_B and X_F. Now notice that the only difference between Y_{j^*} and Z_{j^*} is that the former does not have the edges e for which the only incident edge of e in Y'_{j^*} is (u_X,x_e). However, these edges are only useful for moving between X_B through u_X, and thus does not affect whether or not X_B is separated from X_F. Hence $X_{Y_{j^*}\cup Y_{\text{good}}}$ also separates X_B from X_F. □

Therefore, if we repeat the above refinement process, each time we shrink the capacity of Y_{bad} by a constant factor. So after $O(\log n)$ iterations, we are done finding a desired set of cut edges Y that can route to both B and F with the desired congestion. Moreover, in each iteration, we can determine in parallel which edges in Y_{bad} have become good by simply examining our flow solution. The lemma thus follows. □

We now prove Lemma 5.3.4.

Proof. (Lemma 5.3.4) The proof is essentially the same as Lemma 3.2 of [RST14], with their second partitioning step replaced by ours. Using Lemma 5.3.6 with $\psi=1$, we can find an edge set C such

that X_C separates X_A and X_B in $G'(\mathcal{P})$ and there exists a desired multicommodity flow routing from X_C to X_A as guaranteed by Lemma 5.3.6. Moreover, the total capacity of C is at most twice of the total capacity of B. Using the fact that X_C separates X_A and X_B in $G'(\mathcal{P})$, and that $A \cup B$ induces a 3/4-balanced partition, due to Claim 1 in the Proof of Lemma 3.2 in [RST14], either $A \cup C$ or $B \cup C$ induces a 3/4-balanced partition. If it is $A \cup C$, then we have achieved the second case of the lemma. If it is $B \cup C$, we have achieved the first case of the lemma. □

5.3.3. Recursive Construction of Congestion Approximators

We now show how to recursively construct a congestion approximator for G using our two partitioning steps above. Note that imperatively, we use a lower reweighting factor for the boundary edges in `partition-A` than in `partition-B` ($\log^{-12} n$ vs. $\log^{-4} n$) to avoid a blowup in the congestion when doing the routing "fixing" phase, which is needed because we route on contracted subgraphs rather than vertex-induced subgraphs.

Similar to [RST14], our congestion approximator will also be a hierarchical decomposition tree of the graph. For a tree node corresponding to a subset of vertices $S \subset V$, the tree edge that connects this tree node to its parent will have capacity equal to the total capacity of edges leaving S in G (i.e. the capacity of the cut $(S, V(G) \setminus S)$ in G).

We give the pseudocode of our construction of a congestion approximator of G below. As we highlighted at the beginning of this section, this algorithm implements our Algorithm 5.1.2 modulo being able to solve approximate maximum flows on contracted subgraphs, which we will show how to do in the next section (Section 5.4). During the execution of our algorithm, we do not reweight the graph that we recurse on, but only do reweighting when running the two partitioning steps. We slightly abuse notations by writing a number γ to denote a constant reweighting function that evaluates to γ on every boundary edge. Initially, we call the algorithm on the entire graph with $X = \emptyset$.

Theorem 5.3.9. *The algorithm* `hierarchical-decomp`(G) *has* $O(m \operatorname{polylog}(n))$ *total work and* $O(\operatorname{polylog}(n))$ *depth. Moreover, the output tree* $\mathcal{T} =$ `hierarchical-decomp`(G) *is an* $O(\log^9 n)$*-congestion approximator of* G *with high probability.*

Algorithm 5.3.1 `hierarchical-decomp`$(G(\mathcal{P}))$

Input: Reweighted contracted subgraph $G(\mathcal{P})$, where $\mathcal{P} = (S, 1)$.

Output: A hierarchical tree decomposition tree $\boldsymbol{T}$ of G.

Procedure:

if $G(\mathcal{P})$ only has one uncontracted vertex, plus the supernode u_X **then**
    Return the single uncontracted vertex as our congestion approximator and abort.
Run `partition-A` on $G(\mathcal{P}_A := (S, \log^{-12} n))$ to get a partition Z_1, Z_2 of the uncontracted vertices $V(G(\mathcal{P})) \setminus \{u_X\}$, with F being the inter-cluster edges between Z_1, Z_2.
Run `partition-B` on $G(\mathcal{P}_B := (S, \log^{-4} n))$ with $\psi = \log^{-6} n$ to obtain a set of cut edges Y separating boundary edges B from inter-cluster edges F.
// Recall that B is the set of edges incident to u_X in $G(\mathcal{P}_B)$.
Let L_1 denote the vertices in $V(G(\mathcal{P}))$ that cannot reach any edge in B after the removal of Y, and let L_2 denote the other vertices.
// L_2 are the vertices that can reach B after the removal of Y
Let $\mathcal{Z} := \{\mathcal{Z}_1 := Z_1 \cap L_1, \mathcal{Z}_2 := Z_2 \cap L_1, \mathcal{Z}_3 := Z_1 \cap L_2, \mathcal{Z}_4 := Z_2 \cap L_2\}$ be the partition obtained by taking the intersection of the partitions returned by `partition-A` and `partition-B`.
for $\mathcal{Z}_i \in \mathcal{Z}$ **do**
    Recursively run `hierarchical-decomp` on $G(\mathcal{P}_i)$ to get a tree $\mathcal{T}_i$, where $\mathcal{P}_i = (\{S_i := \mathcal{Z}_i\}, 1)$
Create a new tree node r as the root of our tree, and two other nodes v_{L_1}, v_{L_2} as r's children corresponding to vertex subsets v_{L_1}, v_{L_2} respectively. We also create nodes v_1, v_2, v_3, v_4 corresponding to $\mathcal{Z}_1, \mathcal{Z}_2, \mathcal{Z}_3, \mathcal{Z}_4$ respectively, and let v_1, v_2 be the children of L_1 and let v_3, v_4 be the children of L_2.
For each v_i, let v_i be the root of the tree $\mathcal{T}_i$ that we computed using recursive calls to `hierarchical-decomp`. In other words, the children of L_1 are the roots of $\mathcal{T}_1$ and $\mathcal{T}_2$, and the children of L_2 are the roots of $\mathcal{T}_3$ and $\mathcal{T}_4$.
For each tree node corresponding to a subset of vertices $S \subset V(G)$, weight the tree edge that connects this tree node to its parent by the total capacity of the cut $(S, V(G) \setminus S)$ in G.
return The tree rooted at r constructed above.

Proof. The total work and depth follows easily from the performance of the two partitioning steps above. We thus focus on proving that the returned tree $\mathcal{T}$ is an $O(\log^9 n)$-congestion approximator. Specifically, we show that

1. Any multicommodity flow demands that can be routed in G with congestion 1 can also be routed on the tree with congestion at most 1.
2. Any multicommodity flow demands that can be routed on the tree with congestion 1 can also be routed in G with congestion $O(\log^9 n)$.

Here 1 is clear, since the tree cut induced by each tree edge corresponds to a cut in the original graph with exactly the same capacity, and thus the set of these tree cuts are a subset of the cuts in the original graph. We then prove 2. Consider that there is a demand D_{st} between each vertex pair $s, t \in V(G)$ such that these demands D_{st}'s can be routed simultaneously on the tree with congestion at most 1. We then describe a routing scheme to show that they can also be simultaneously routed in G with congestion $O(\log^9 n)$. Our routing scheme will be invoking an essentially same routing routine in [RST14], plus an additional routing *fixing* phase, which is necessary due to the fact that we work in contracted subgraphs rather than vertex-induced subgraphs.

To illustrate a routing between s and t of flow value D_{st}, we let each of s, t send out a message of size D_{st}. We will thus use the terms *flow* and *message*, as well as *message passing* and *flow routing*, interchangeably. We will move these messages up along the hierarchical decomposition tree by repeatedly moving the messages from the boundary edges of the current cluster to the boundary edges of the parent cluster. If at some point, to some edge, we have routed the same amount of flow from s and t, then we can construct a routing from s to t of this flow amount by reversing the message trail of t; whenever this happens in our routing process, we discard these paired-up messages and never consider them in subsequent routing steps. Crucially, we move these messages for different s, t pairs *simultaneously*.

We will specify this message passing process inductively. Note that in each recursive call of the tree construction, we always do two steps of partitioning and obtain a two-level tree before recursing on the leaves of the two-level tree corresponding to smaller clusters. Let us call the two levels in each single recursive call a *level block*. We will route the messages level block by level block. For each

level block with root being $S = V(G) \setminus X$, we specify a routing in the graph $G'(\mathcal{P} = (S, 1))$, the subdivision of the contracted subgraph. When we say we route some flow to an edge in $G(\mathcal{P})$ (the contracted subgraph without subdivision), we mean we route the flow to the split vertex of that edge. We will maintain the following invariant:

> **Invariant:** *After we are done with the routing for a level block with root being $S = V(G) \setminus X$, all messages originating from vertices in S either have been discarded, or reside on the boundary edges (i.e. the ones that go between S and u_X).* $(\star)$

We will also do the routing "fixing" inductively. Specifically, in each inductive step, even before we specify the routing, we first fix the routing for each smaller cluster $\mathcal{Z}_i$ that we specified in the smaller contracted subgraph $G'(\mathcal{P}_i = (\mathcal{Z}_i, 1))$, so that they become routings in the bigger contracted subgraph $G'(\mathcal{P} = (S, 1))$. Only after the fixing is done for each $\mathcal{Z}_i$ do we proceed with the routing of the messages in $G'(\mathcal{P})$.

As a result, our inductive step consists of two phases, namely a *fixing phase*, and a *routing phase*, where the routing phase will be essentially the same as in [RST14]. We now fully describe our inductive routing scheme below, as well as analyze the congestion caused. Notice that, throughout the proof, we always analyze the congestion of edges with respect to their *original* capacities in G (without considering the reweighting factors that we use in contracted subgraphs). We will highlight all the congestion we calculate along the way of presenting our routing scheme. Since we route in subdivision graphs, we will need to distinguish, for a boundary edge e, the congestion on its split edge incident on u_X, and the congestion on its other split edge incident on S. We will call the former outer congestion, and the latter inner congestion.

Note that for our goal of proving the obtained tree is a congestion approximator, it suffices to prove the existence of such a low-congestion routing. We will then plug in the tree into Sherman's algorithm to compute approximate maximum flows.

Base Case. Initially, as the base case of the inductive routing process, for each demand D_{st}, s and t will each distribute the total amount of flow D_{st} uniformly to their outgoing edges - a.k.a. the boundary edges of these singleton clusters - with each edge getting flow proportional to its capacity.

Thus we have established the invariant ($\star$).

This routing causes congestion at most 2 on the edges of G, since a node cannot send or receive a total flow amount greater than its weighted degree given that the demands are routable with congestion 1, and an edge is incident on two vertices.

Inductive Step: Fixing Phase. For $i = 1, 2, 3, 4$, given a routing in $G'(\mathcal{P}_i)$, we aim to obtain another routing in the bigger contracted subgraph $G'(\mathcal{P})$ that route the same demands. Recall that S denotes the set of nodes in consideration, i.e., $S = V \setminus X = \mathcal{Z}_1 \cup \mathcal{Z}_2 \cup \mathcal{Z}_3 \cup \mathcal{Z}_4$. Suppose first we take the exact same given routing in $G'(\mathcal{P}_i)$ and simply expand (namely, undo the contractions of) the vertices in $S \setminus \mathcal{Z}_i$, obtaining a routing in $G'(\mathcal{P})$. Due to the expanding we have done, we may have created extra deficits and excesses on the vertices in $(S \setminus \mathcal{Z}_i) \cup \{u_X\}$, which we now have to fix. To this end, it suffices to give for each demand a routing with the deficits/excesses on $(S \setminus \mathcal{Z}_i) \cup \{u_X\}$ switched. These routings should be simultaneously achievable with low congestion.

Let $\alpha_i \geqslant 0$ be the maximum outer congestion on the boundary edges of $G(\mathcal{P}_i = (\mathcal{Z}_i, 1))$, and let $\alpha := \max\{\alpha_1, \alpha_2, \alpha_3, \alpha_4\}$. Let $\gamma_i \geqslant 0$ be the maximum inner congestion on the boundary edges of $G(\mathcal{P}_i = (\mathcal{Z}_i, 1))$, and let $\gamma := \max\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$.

Claim 5.3.10. *We can fix the routings in $G(\mathcal{P}_i)$'s so that the become routings in $G(\mathcal{P})$ with the same demand while causing outer congestion of boundary edges $O(\alpha \log^{-3} n)$, inner congestion of boundary edges $O(\alpha \log n)$, and congestion on edges inside S $O(\alpha \log^{8} n)$.*

Proof. For $\mathcal{Z}_3, \mathcal{Z}_4$, we fix the extra deficits/excesses by a routing from $Y \setminus B$ to B, followed by a mixing to uniform (with each edge getting flow proportional to its capacity) over B. The routing causes outer congestion of boundary edges $O(\alpha \log^{-9} n)$, inner congestion of boundary edges $O(\alpha \log^{-3} n)$, and congestion of edges inside S $O(\alpha \log n)$. The mixing causes congestion $O(\alpha \log^{-3} n)$. The last congestion is because each edge in B receives $O(\alpha \log^{-3} n)$ times its capacity units of flow,

For $\mathcal{Z}_1, \mathcal{Z}_2$, we fix the extra deficits/excesses by a routing from Y to F, followed by a mixing to uniform (with each edge getting flow proportional to its capacity) over F. The routing causes outer congestion of boundary edges $O(\alpha \log^{-5} n)$, inner congestion of boundary edges $O(\alpha \log n)$, and congestion of edges inside S $O(\alpha \log^{5} n)$. The mixing causes congestion on boundary edges

$O(\alpha \log^{-4} n)$, and congestion on edges inside S $O(\alpha \log^{8} n)$. □

Inductive Step: Routing Phase. This phase is essentially the same as in [RST14], except that we do the routing in the contracted subgraph $G'(\mathcal{P})$ rather than in the vertex induced subgraph, and therefore we also have to analyze the congestion on the boundary edges B.

For the routing phase, we divide the messages originating in S that haven't yet been discarded (thus still need to be rounted) into following types:

- **Type 1** The messages corresponding to demands D_{st}'s for which both s, t lie in $L = \mathcal{Z}_1 \cup \mathcal{Z}_2$ (vertices not reachable from B after removing Y).
- **Type 2** The messages corresponding to demands D_{st}'s for which one of s, t lies in L and the other lies outside of L.
- **Type 3** The messages corresponding to demands D_{st}'s for which both s, t lie outside L (but at least one of s, t lies in $R = S \setminus L$, as otherwise they are not considered in this level block).

Recall that, by the invariant $(\star)$ that we maintain, all three types of messages reside on the boundary edges of the smaller clusters $\mathcal{Z}_i$'s. Note that, by the construction of the $\mathcal{Z}$'s, the boundary edges of clusters $\mathcal{Z}_1, \mathcal{Z}_2$ are subsets of $F \cup Y$, while the boundary edges of $\mathcal{Z}_3, \mathcal{Z}_4$ are subsets of $Y \cup B$. The goal of our inductive routing step is as follows:

- **Goal 1** For **Type 1** messages, we show that we can simultaneously pair up the messages from s and the ones from t.
- **Goal 2** For **Type 2** and **Type 3** messages, we show how to route them to the boundary edges B of the bigger cluster S.

Here **Goal 1** means that we can discard all **Type 1** messages afterwards, since by doing so we have already found a simultaneous routing of the corresponding demands D_{st}'s; whereas **Goal 2** means that after our routing, **Type 2** and **Type 3** messages always reside on the boundary edges of S. These together imply that we have maintained our invariant $(\star)$.

We now describe how we achieve these goals, as well as analyze the congestion caused. Throughout, let $\beta_i \geqslant 0$ be the maximum ratio of the amount of flow received to the (original) edge capacity over

the boundary edges of $\mathcal{Z}_i$, when we finish routing in $G(\mathcal{P}_i)$. Then let $\beta = \max\{\beta_1, \beta_2, \beta_3, \beta_4\}$.

Inductive Step: Routing Type 1 Messages. By our invariant ($\star$), **Type 1** messages reside on the boundary edges of $\mathcal{Z}_1 \cup \mathcal{Z}_2$, which are $Y \cup F$. We first route **Type 1** messages that reside on $Y \setminus F$ to F. By the guarantee of `partition-B`, this can be done with outer congestion of boundary edges $O(\beta \log^{-5} n)$, inner congestion of boundary edges $O(\beta \log n)$, and congestion of edges inside S $O(\beta \log^5 n)$.

After the routing, all **Type 1** messages reside on F, and the flow that each edge in F carries is at most $2\beta + O(\beta \log^5 n) = O(\beta \log^5 n)$ of its capacity. We then simultaneously mix each of **Type 1** messages uniformly over F, in the sense that each edge in F gets an amount proportional to its capacity. After the mixing, we have successfully paired up all **Type 1** messages and thus can discard them all.

This mixing causes congestion on boundary edges $O(\beta \log^{-4} n)$, and congestion on edges inside S $O(\beta \log^8 n)$. Combined this with the congestion we get in the $Y \setminus F \to F$ routing, the added outer and inner congestion of B in $G(\mathcal{P})$ is at most $O(\beta \log^{-4} n)$ and $O(\beta \log n)$, whereas the added congestion on edges inside S in $G(\mathcal{P})$ is at most $O(\beta \log^8 n)$.

Inductive Step: Routing Type 2, Type 3 Messages. By our invariant ($\star$), both **Type 2** and **Type 3** messages reside on the edges in $Y \cup B \cup F$. We first route the messages that reside on $F \setminus Y$ to Y. This routing consists of first mixing the message uniformly on F and then reversing the routing from Y to a uniform distribution over F, whose existence is guaranteed by `partition-B`. After the routing, edges in $F \setminus Y$ carry no flow on them, and each edge in Y carries flow that is at most $2\beta + 1$ of its capacity in $G(\mathcal{P})$.

By a similar analysis as in the routing of **Type 1** messages, the routing can be done with outer congestion of boundary edges $O(\beta \log^{-5} n)$, inner congestion of boundary edges $O(\beta \log n)$, and congestion of edges inside S $O(\beta \log^5 n)$.

We then route message on $Y \setminus B$ to B. The routing causes outer congestion of boundary edges $O(\beta \log^{-9} n)$, inner congestion of boundary edges $O(\beta \log^{-3} n)$, and congestion of edges inside S $O(\beta \log n)$. After the routing, each edge in B carries flow that is at most $\beta + (2\beta + 1)O(\log^{-3} n) =$

$(1 + O(1/\log^3 n))\beta$ of its capacity in $G(\mathcal{P})$.

Total Congestion Analysis. We first obtain an upper bound on β, the total amount of flow carried by a boundary edge divided by its capacity in G. Initially, we have $\beta = O(1)$ in our base case. Then in each inductive step, β grows by $1 + O(1/\log^3 n)$, as we have analyzed above when routing **Type 2**, **Type 3** messages, which is the only place where we route flows to B. Since the depth of the tree is $O(\log n)$, we have $\beta = O(1)$ throughout.

We next obtain an upper bound on α, the total outer congestion of any boundary edge, and γ, the total inner congestion of any boundary edge, In the fixing phase, the total outer congestion and inner congestion added on a boundary edge is $O(\alpha_i/\log^3 n)$ and $O(\alpha \log n)$ respectively. In the routing phase, the total outer and inner congestion added on a boundary edge is $O(\beta \log^{-4} n)$ and $O(\beta \log n)$. Therefore, by a simple induction and the fact that the tree has $O(\log n)$ depth, the total outer and inner congestion accumulated over descendant routing steps on a boundary edge is bounded by $O(1)$ and $O(\log^2 n)$, respectively.

Finally, in the routing phase and the fixing phase, the total congestion added on the edges inside S is bounded by $O(\beta \log^8 n) + O(\alpha \log^8 n) = O(\log^8 n)$. Since each edge appears $O(\log n)$ times as inside S, the total congestion accumulated is $O(\log^9 n)$. This finishes the proof of the theorem. □

5.4. Extraction of Congestion Approximators

We finally describe the implementation details of the approximate maximum flow oracle $\mathcal{A}$ in Section 5.3 whose performance guarantees are summarized in Proposition 5.3.2. We would like to remind the reader that this oracle, at its heart, utilizes Sherman's framework to compute approximate maximum flows which requires as input polylog n-congestion approximators for the flow instance upon which it is invoked. While we defer the PRAMimplementation details of Sherman's algorithm to Appendix C.3, we discuss in this section, three important routines for constructing congestion approximators required for the contracted subgraphs generated by our new framework for constructing high-quality congestion approximators. These are (i) (partially) compressing the global congestion approximator to obtain one for the contracted subgraphs, (ii) obtaining a near-linear work, low-depth implementation of Sherman's algorithm when given as input these (partially) com-

pressed congestion approximators, and (*iii*) constructing congestion approximators for contracted subgraphs with an arbitrarily attached super-source and super-sink. We would crucially like to remind the reader that at all points in our overall approach, we build and maintain congestion approximators for *subdivision graphs* and *contracted subdivision graphs*, as these are precisely the flow instances upon which the max-flow oracle is invoked.

5.4.1. Compressing Congestion Approximators

We start by describing the `ca-contraction` subroutine of Algorithm 5.1.2 that given a contracted subgraph and a congestion approximator for a much larger graph, compresses it to have size proportional to that of the contracted subgraph. Specifically, we are given as input a $O(\text{polylog}\, n)$-congestion approximator R for some larger graph $G = (V, E, c)$ along with a subset $S \subset V$ of k vertices that have the following special property: all edges that leave the set S are well-linked in G. We prove such a property for the graphs we encounter:

Claim 5.4.1. *In any $G(\mathcal{P} = (S, 1))$ with $X = V(G) \setminus S$ and S being a cluster corresponding to a leaf of a level block of the tree constructed by `hierarchical-decomp`, the boundary edges incident on u_X are $\Omega(1/\log^9 n)$-well-linked in G.*

Proof. Note that in $G(\mathcal{P})$, these edges are 1-well-linked. We then consider the multicommodity flow routing between them in $G(\mathcal{P})$. We now fix this routing using our fixing step with performance guaranteed by Claim 5.3.10 in a bottom-up manner until we have converted the routing into a valid one in G with the same demands. Then by Claim 5.3.10 the total congestion is $O(\log^9 n)$, implying the claim. □

Our goal is to output a $O(\text{polylog}\, n)$-congestion approximator of size $O(k \log n)$ for the contracted subgraph $G(S)$ where $V \setminus S$ is compressed into a single node[17], and the aforementioned well-linkedness property will be crucial in achieving it. This compression step is critical in order to achieve near-linear work; while the original congestion approximator R of G is also a $O(\text{polylog}\, n)$-congestion approximator for the contracted graph $G(S)$, its size may be very large relative to that

[17] with some edges properly reweighted; see Section 5.3.3. Since this does not affect the algorithm or analysis of this section, we assume this reweighting has been done prior to the call to `ca-contraction`.

of $G(S)$, and naively using it for computing approximate max-flows in $G(S)$ would substantially blow up the total work. This subroutine precisely addresses this issue by reducing the size of the congestion approximator without degrading its quality substantially.

If we were simply to contract into a *single node* all the nodes in R which represent only subsets of $V \setminus S$, the resulting graph would not be a tree, and so we would not be able to use it for computing max-flows with Sherman's algorithm. As such, we must use an alternative approach to shrink the size of R that preserves the tree structure. We accomplish this with the following algorithm, which uses all three properties guaranteed by the transformation in Section 5.2.2: R is a hierarchical decomposition and also a binary tree of depth $O(\log n)$. As R is a hierarchical decomposition, for all $v \in S$, there is a leaf of R corresponding to the set containing only v. Assign a node weight of 1 to these leaves, and assign a weight of 0 to all other nodes in R. Then, run the subtree sum algorithm of Theorem C.1.4 and contract all subtrees whose subtree sum is 0. As before, for a graph G and set $S \subseteq V(G)$, let $G(S)$ be G with $V(G) \setminus S$ contracted into a single node. We use the fact that R is binary and has $O(\log n)$ depth to bound the size of resulting tree.

Algorithm 5.4.1 `ca-contraction`(R, S)

Input: α-congestion approximator R (for a graph $G = (V, E, c)$), which is a hierarchical decomposition and a binary tree of depth $O(\log n)$; a subset $S \subset V$ of $|S| = k$ uncontracted vertices.
Output: Tree R' with $O(k \log n)$ nodes that is an $(\alpha \cdot \text{polylog}\, n)$-congestion approximator for $G(S)$.
Procedure:

1: Assign a node weight of 1 to each leaf u of R such that $P_u = \{v\}$ for some $v \in S$, where $P_u \subseteq V$ is the partition of vertices the node u corresponds to in the hierarchical decomposition.
2: Assign a node weight of 0 to all other nodes of R.
3: Compute the subtree sums with respect to these weights using the algorithm of Theorem C.1.4.
4: From top-down, contract each subtree whose sum is 0 into a single supernode.

It is important to note that this algorithm does *not* require all leaves of the input R to correspond to single vertices, as is the case in a standard hierarchical decomposition tree. In Algorithm 5.1.2, we contract congestion approximators that themselves have been contracted from a previous tree, and so not all leaves may correspond to single vertices.

Lemma 5.4.2. *The* `ca-contraction`(R, S) *subroutine has depth* $O(\log n)$ *and* $O(|R| \log n)$ *total work, and outputs a tree* R' *with* $O(k \log n)$ *nodes.*

Proof. By Theorem C.1.4 and the fact that R has $O(\log n)$ depth, the algorithm has $O(\log n)$ depth and $O(|R| \log n)$ total work. To bound the size, first note by the fact that R is a hierarchical decomposition, if $u_1, \ldots, u_q$ are the nodes of R at some level i, the $P_{u_1}, \ldots, P_{u_q}$ are a partitioning of V. So, at each level of R, there can be at most k nodes u such that $P_u \cap S \neq \emptyset$, and so at most k nodes can remain uncontracted at each level after contraction. Moreover, since R is a binary tree, every uncontracted node can have at most two supernodes as children, and, by definition, all supernodes are leaves. Since R has depth $O(\log n)$, it thus follows that the size of the resulting tree R' is $O(k \log n)$, as desired. □

It thus remains to show that the contracted tree R' can still be used to route flow with low congestion in the graph with $V \setminus S$ contracted into a single node. R' is constructed by contracting some subtrees of R; call the root of these contracted subtrees (along with any remaining leaves corresponding to single nodes from $V \setminus S$) $u_1^*, \ldots, u_q^*$ and let $T_i = P_{u_i^*}$, ordering arbitrarily. So, $T_1 \cup \ldots \cup T_q = V \setminus S$. Let $\bar{G}(S)$ be the graph with each T_i contracted to a single node, but these contracted nodes are *not* further contracted. Since R' is constructed by contracting the nodes corresponding to each T_i in R, R' is a α-congestion approximator for $\bar{G}(S)$ (where α is the congestion achieved by R for routing on G). Our goal is thus to show that R' can be used as an $(\alpha \cdot \operatorname{polylog} n)$-congestion approximator for $G(S)$.

Let x be the contracted node (i.e. the node formed by contracting $V \setminus S$ in G) in $G(S)$, and suppose we are given a demand vector b on $G(S)$. Furthermore, for each $i \in [q]$, let x_i be the node in $\bar{G}(S)$ formed by contracting T_i. If we were able to efficiently split the demand b_x into demands for $x_1, \ldots, x_q$ without inducing much additional congestion, then we would be able to use R' as a congestion approximator for $G(S)$. Namely, we first convert the demand on $G(S)$ into the corresponding demand on $\bar{G}(S)$, and then use R' to route the flow on $\bar{G}(S)$ which is also a routing on $G(S)$ (by replacing any x_i with x). So, it remains to show that we may indeed split the demand b_x into demands $b'_{x_1}, \ldots, b'_{x_q}$ such that $b_x = \sum_{i \in [q]} b'_{x_i}$ and b' can be routed on $\bar{G}(S)$ with low congestion. This follows from the fact that we are guaranteed the set of edges leaving S in G are $\Omega(1/\log^9 n)$-well-linked (Claim 5.4.1), and is formalized in the following lemma. Importantly, since R' remains an α-congestion approximator for $\bar{G}(S)$, for any $Q \subseteq S$, `ca-contraction`(R', Q)

is also an α-congestion approximator for $\bar{G}(Q)$[18]. As such, the error does not accumulate when repeatedly contracting a congestion approximator, as we do in Algorithm 5.1.2.

Lemma 5.4.3. *Let b be a demand vector on $G(S)$ which can be satisfied with congestion 1. Let $W_x = \sum_{(u,x)\in E} c((u,x))$ be the sum of capacities of edges incident on x in $G(S)$, and define W_{x_i} similarly for each x_i in $\bar{G}(S)$. Then, with b' the demand vector on $\bar{G}(S)$ such $b'_u = b_u$ for all $u \in S$ and $b'_{x_i} = W_{x_i} b_x / W_x$, b' can be satisfied with congestion $O(\log^9 n)$.*

Proof. Consider some flow f in $G(S)$ which satisfies b with congestion 1. Let $S = \{(u,x) \mid u \in S\}$ be the set of edges leaving S in $G(S)$, and similarly let $S_i = \{(u,x_i) \mid u \in S\}$ be the set of edges incident on x_i in $\bar{G}(S)$. Note that for each $(u,x) \in S$, there exists a corresponding edge (u,x_i) (for some x_i) in $\bar{G}(S)$ with the same capacity, by the construction of x and $x_1, \ldots, x_q$. So, we may convert f into a flow on $\bar{G}(S)$; call this f'. Define the flow vector b^* such that $b^*_{x_i}$ is the net incoming flow to x_i induced by f', and $b^*_u = b_u$ for all $u \in S$. f' has congestion 1, by assumption on b, and satisfies the demand vector b^*, so it follows that b^* can be satisfied in $\bar{G}(S)$ with congestion 1 as well. As S is $\Omega(1/\log^9 n)$-well-linked for each S on which we call `ca-contraction` in Algorithm 5.1.2 (as shown in Claim 5.4.1), each S_i is as well by Proposition 2.5.9. Thus, by definition of well-linked and the fact that b and b^* differ only on the x_i, it follows that b' can be satisfied on $\bar{G}(S)$ with congestion $O(\log^9 n)$. □

There is one last point to check: that this splitting procedure can be implemented in $O(\log n)$ depth and $O(|R|)$ work. This is not as simple as it may initially seem: since there could be potentially $\Theta(k \log n)$ contracted nodes x_i in $\bar{G}(S)$, naively summing the weight on all edges could result in $\Omega(k^2)$ work. Fortunately, given access to the contracted graph $G(S)$ (which are computed in Algorithm 5.1.2), we can use subtree sums to implement the demand splitting efficiently. Iterate through these edges, and for each $(u,v) \in B$, where B is the boundary edges of $G(S)$, such that $u \in S$ and $v \in V \setminus S$, assign a node weight of $c((u,v))$ to leaf of R which corresponds to v. Assigning these weights can be implemented in work $O(|B|)$; this suffices to prove that Algorithm 5.1.2 requires only $\tilde{O}(m)$ total work and does not affect the runtime of `ca-contraction`. Once the node weights have

[18]We slighly abuse notation to refer to $\bar{G}(Q)$ as the graph with the same components contracted as in the tree output of `ca-contraction`(R', Q)

been assigned, compute the subtree sums for each node in R, which can be done in $O(\log n)$ depth and $O(|R|)$ work by Theorem C.1.4. The sum W_{x_i} used in Lemma 5.4.3 is then exactly the subtree sum of the node corresponding to the set T_i, allowing us to correctly distribute the demands.

5.4.2. Implementing Sherman's Algorithm on the Contracted Subgraph

Next, we discuss how we achieve a linear-work, low-depth implementation of Sherman's algorithm on the contracted subgraph $G(S)$, where $S \subset V$ is the set of $|S| = k$ uncontracted vertices, with all other vertices $V \setminus S$ being contracted into a single super-vertex x. We encourage the reader to familiarize themselves with the vanilla implementation of Sherman's algorithm outlined in Appendix C.3 to obtain a better understanding of the discussion that follows.

Recall from the preceding section, that we are given access to an α'-congestion approximator R' for a slightly larger graph $\bar{G}(S)$, where the vertices $V \setminus S$ have been partitioned and contracted into multiple super-vertices $x_1, \ldots, x_q$. In order to use this congestion approximator R' for our desired contracted graph $G(S)$, we need to translate demands on vertices in $G(S)$ into demands on vertices in $\bar{G}(S)$. The optimization problem within the `AlmostRoute` subroutine of [She13b] therefore becomes a minimization problem over the new congestion potential

$$\phi(f) = \mathsf{lmax}(C^{-1}f) + \mathsf{lmax}(2\alpha' R' P(Bf - b)),$$

where B is the vertex-edge incidence matrix for $G(S)$, and P is a linear operator that projects any demands b supported over vertices of $G(S)$ to demands b' supported over vertices of $\bar{G}(S)$ as described in Lemma 5.4.3. Due to the fact that R' is a $O(k \log n)$ size, $O(\log n)$ depth tree, evaluating this potential is easy; its computation remains unchanged from the vanilla case described in Appendix C.3. The only major challenge here is efficiently computing the derivatives of this new potential, specifically, the derivative of the second term

$$\phi_2(f) := \mathsf{lmax}(2\alpha' R' P(Bf - b)).$$

First, observe that the operation $P \cdot B$ effectively constructs a new vertex-edge incidence matrix[19] B' for $\bar{G}(S)$ from the incidence matrix B in the following way: for every edge $e = (u_1, u_2)$ in $G(S)$ with only uncompressed vertices $u_1, u_2 \in S$ as its endpoints, this operation replicates the edge exactly in B', i.e. $B'_{v,e} = B_{v,e} \in \{-1, 1\}$, for $v \in \{u_1, u_2\}$ and 0 otherwise. However, for any edge $e = (u, x)$ (or (x, u)) with one of its endpoints being the contracted vertex x, this operation splits the edge e into fractional copies $e_1 = (u, x_1), \ldots, e_q = (u, x_q)$ with copy e_i having fractional value $\rho_{x_i} := W_{x_i}/W_x$ as defined in Lemma 5.4.3. i.e. for each $i \in [q]$, $B'_{x_i,e_i} = \rho_{x_i} \cdot B_{x,e}$, $B'_{u,e_i} = \rho_{x_i} \cdot B_{u,e}$, and 0 otherwise. Note that we never consider edges going between two partially contracted vertices x_i, x_j, as they are absent in $G(S)$. While we do not explicitly compute this new incidence matrix B' as doing so naively might exceed our linear (in size of $G(S)$) work requirement, it will serve as a useful intermediate object for analyzing the gradients.

Now let $\mathcal{I}'$ be the set of all cuts considered by our congestion approximator R', and for any cut $i = (S_i, \overline{S_i}) \in \mathcal{I}'$, let $y_i = 2\alpha'[R'P(Bf - b)]_i$ be the congestion induced by the residual demands across cut i. We have that for any edge $e \in G(S)$, the partial derivative

$$\frac{\partial \phi_2(f)}{\partial f_e} = \sum_{i \in \mathcal{I}'} \frac{\partial \phi_2(f)}{\partial y_i} \cdot \frac{\partial y_i}{\partial f_e} = \sum_{i \in I'} \frac{\exp(y_i) - \exp(-y_i)}{\exp(\phi_2(f))} \cdot \frac{2\alpha' B'_{S_i,e}}{c(S_i, \overline{S_i})},$$

where $c(S_i, \overline{S_i})$ is the capacity of cut $i = (S_i, \overline{S_i})$ in $\bar{G}(S)$ considered in our congestion approximator R', and (with some abuse of notation) $B'_{S_i,e}$ represents the total "fraction" of the edge e crossing the cut $(S_i, \overline{S_i})$; for edges $e = (u_1, u_2)$ with only uncompressed vertices as its endpoints, this quantity is $B'_{S_i,e} = \sum_{v \in S_i} B'_{v,e} = \sum_{v \in S_i \cap S} B_{v,e}$, and for edges $e = (u, x)$ (or (x, u)) with one of its endpoints being the compressed vertex, this quantity is $B'_{S_i,e} = \sum_{j \in [q]} B'_{S_i,e_j} = \sum_{j \in [q]} \sum_{v \in S_i} B'_{v,e_j} = \sum_{v \in S_i \cap S} B_{v,e} + \sum_{v \in S_i \setminus S} \rho_v \cdot B_{x,e}$, where the set $\{e_j\}_{j \in [q]}$ correspond to the fractional copies of edge e constructed by the operation $P \cdot B$.

In order to efficiently compute this gradient, we shall again exploit the fact that R' is represented by a rooted hierarchical decomposition tree T' of size $O(k \log n)$ and depth $O(\log n)$. To do so, we use the same node-potential trick as in Appendix C.3: for any internal node j in T' (which in turn

[19] Note that this new vertex-edge incidence matrix B' may not be the same as the actual incidence matrix of $\bar{G}(S)$. However, this is how it effectively appears to Sherman's algorithm when invoked with R' as its input.

corresponds to a cut $(S_j, \overline{S_j})$), we define the node potential π_j as

$$\pi_j := \sum_{i \in T'_{j,r}} \frac{\exp(y_i) - \exp(-y_i)}{\exp(\phi_2(f))} \cdot \frac{2\alpha'}{c(S_i, \overline{S_i})},$$

where $T'_{j,r}$ denotes the path in T' from node j to the root r of T'. Now observe that, following an identical calculation as in Appendix C.3, the gradient of $\phi_2(f)$ w.r.t. the flow f_e on any edge $e = (u_1, u_2)$ with only uncompressed vertices as its end-points remains unchanged from the vanilla case.

$$\frac{\partial \phi_2(f)}{\partial f_e} = \sum_{i \in T'_{u_1,u_2}} \frac{\exp(y_i) - \exp(-y_i)}{\exp(\phi_2(f))} \cdot \frac{2\alpha B'_{S_i,e}}{c(S_i, \overline{S_i})} = \pi_{u_2} - \pi_{u_1},$$

where T'_{u_1,u_2} denotes the unique path between u_1, u_2 in the tree T'. For any edge $e = (u, x)$ (or (x, u)) with one of its end points being the contracted vertex x that is in turn partitioned into fractional edges $e_1 = (u, x_1), \ldots, e_q = (u, x_q)$ in B', observe that this gradient

$$\begin{aligned}\frac{\partial \phi_2(f)}{\partial f_e} &= \sum_{j \in [q]} \sum_{i \in T'_{u,x_j}} \frac{\exp(y_i) - \exp(-y_i)}{\exp(\phi_2(f))} \cdot \frac{2\alpha B'_{S_i,e_j}}{c(S_i, \overline{S_i})} \\ &= \sum_{j \in [q]} \rho_{x_j}(\pi_{x_j} - \pi_u) = \left(\sum_{j \in [q]} \rho_{x_j} \pi_{x_j}\right) - \pi_u,\end{aligned}$$

where the final equality follows from the fact that $\sum_{j \in [q]} \rho_{x_j} = 1$. We can easily compute all these node potentials π_v (through a prefix sum over an Eulerian tour of the congestion approximator tree T' starting at its root) and precompute the quantity $\sum_{j \in [q]} \rho_{x_j} \pi_{x_j}$ in the PRAMmodel with $O(|R'|) = O(k \log n)$ work and $O(\log n)$ depth. Since this is all we need, namely, be able to efficiently evaluate congestions and compute gradients over the supplied congestion approximator, we have an efficient implementation of Sherman's algorithm on the contracted subgraph $\bar{G}(S)$.

5.4.3. Computing a Congestion Approximator for $G \cup \{s, t\}$

The final requirement is the following: in the cut-matching game of Section C.2, we need to compute $(1 - 1/\operatorname{polylog} n)$-approximate max flow on G with the addition of a source s and sink t which are arbitrarily connected to G. As such, we must convert our α-congestion approximator R for G into

a $O(\alpha \text{ polylog } n)$-congestion approximator R' for $G \cup \{s,t\}$. In this section, we specify how this is achieved.

Lemma 5.4.4. *Let R be a α-congestion approximator for G which is a hierarchical decomposition, and let s,t be two additional vertices connected arbitrarily (and with arbitrary capacities) to G. Then, there is a* polylog k *depth, $\tilde{O}(k)$ work PRAM algorithm which computes a $O(\alpha \text{ polylog } n)$-congestion approximator R' for $G \cup \{s,t\}$, where $k = |R|$.*

Note that $R \cup \{s,t\}$, with s,t connected to the leaves of R corresponding to their neighbors, can be used to route any feasible flow on $G \cup \{s,t\}$ with congestion α, since R is an α-congestion approximator for G. However, $R \cup \{s,t\}$ is not a tree, and thus cannot be readily plugged into Sherman's algorithm to compute maximum flows in $G \cup \{s,t\}$. It thus remains to obtain a congestion approximator (that is a tree) for $G \cup \{s,t\}$, which boils down to computing a hierarchical decomposition (as in [RST14]) for $R \cup \{s,t\}$. The algorithm is quite similar to the procedure of Section 5.2.2, but with a few key modifications to account for the nodes s and t.

Recall from Section 5.2.2 that it suffices to implement the two partitioning steps Partition A and Partition B; we use the same definitions as in Section 5.2.2.

Lemma 5.4.5. *Partition A[20] can be implemented in $O(\log |Q|)$ depth and $\tilde{O}(|Q|)$ work on any subset Q of nodes of $R \cup \{s,t\}$.*

Proof. We use the following procedure:

1. If $s \in Q$, then output the partition $(\{s\}, Q \setminus \{s\})$
2. If $t \in Q$ and $s \notin Q$, then output the partition $(\{t\}, Q \setminus \{t\})$
3. If $s \notin Q$ and $t \notin Q$, then Q induces a subtree of R, and we run the procedure of Lemma 5.2.10

The depth and work are immediate from Lemma 5.2.10. It remains to show that the edges between the partition that is output are 1-well-linked. If $s \in Q$ or $t \in Q$, then all the edges between the outputted partitions are incident on s or t and are thus 1-well-linked. If $s \notin Q$ and $t \notin Q$, then the edges between the outputted partitions are 1-well-linked by Lemma 5.2.10. □

[20] If $s \in Q$ or $t \in Q$, the partitions are not balanced, which does not meet the exact definition of Partition A. However, there are only be 2 such paritionings, and so this can only increase the depth of the final tree by 2.

For Partition B, we only need to apply it on partitions without s or t, or on partitionings where all boundary edges are incident on either s or t, by the construction of Partition A. As such, Partition B reduces to finding the (exact) min-cut on a tree with a source and sink added, exactly as in Section 5.2.2. So, we may use the algorithm of Appendix C.4 to compute Partition B and the analysis follows from Lemma 5.2.11.

The proof of Lemma 5.4.4 is then essentially identical to the proof of Lemma 5.2.12.

5.5. Parallel Flow Decomposition by Shortcutting

We describe our PRAMflow decomposition subroutine in this section, and we will begin by specifying the relevant notation. Consider an s-t flow f specified by a weighted, directed graph $H = (V, E, f)$ containing a source vertex $s \in V$, and sink vertex $t \in V$, with the flow on any edge given by $f : E \to \mathbb{R}^+$. Note that this flow network H is restricted to only the subset of edges that carry *positive* flow, and will be iteratively updated by our algorithm as we make progress towards our flow-decomposition objective. Moreover, while the initial graph H specifying the flow f does not contain any parallel edges, such edges will inevitably end up being created in our flow-decomposition process. Therefore in this section, we will more generally deal with *multigraphs*, whose edges are assumed to be uniquely indexed. Let $|f| := \sum_{(s,v)\in E} f_{(s,v)} - \sum_{(v,s)\in E} f_{(v,s)}$ be the value of the s-t flow. Lastly, we use $S = N^H_{\text{out}}(s)$ to denote the out-neighbors of s corresponding to the "source-side" vertices, and $T = N^H_{\text{in}}(t)$ to denote the in-neighbors of t corresponding to the "sink-side" vertices.

Our goal is to determine how much flow in f is routed between each pair $x \in S, y \in T$. We do so by computing *pairwise* demands[21] $d : S \times T \to \mathbb{R}^+$ such that $\|d\|_1 = |f|$ and d can be routed in H exactly with f being the edge-capacity constraints, and as in the parallel setting, this objective more easily admits a small work and low-depth implementation. We will build a DAG data structure that implicitly encodes the necessary information.

Definition 5.5.1 (Flow Decomposition DAG)**.** An ℓ-layered directed graph $\mathcal{D}$ is a flow decomposition DAG of an s-t flow f specified by a weighted, directed graph H iff

[21]For the objective of computing pairwise demands, we assume that the source and sink-side vertices are non-overlapping, i.e. $S \cap T = \emptyset$, since this will always be the case in our application which is the implementation of the cut-matching game. The parallel flow-decomposition result however, is more generally applicable.

1. Each node of $\mathcal{D}$ corresponds to a directed edge (not necessarily in H) between two vertices u, v in V associated with a flow value $h_{(u,v)} \in \mathbb{R}^+$. We use (u, v, h) to represent this node in $\mathcal{D}$, dropping the subscript (u, v) in the flow value $h_{(u,v)}$ when it is unambiguous for ease of exposition. Note that there can be multiple nodes of the form $(u, v, h_{(u,v)})$ at the same layer of $\mathcal{D}$ due to the existence of parallel (u, v) edges, each with potentially different flow values $h_{(u,v)}$. We assume that all nodes in $\mathcal{D}$ are uniquely indexed to avoid ambiguity.
2. The nodes (u, v, h) at the lowest layer 1 do not have any predecessors. In particular, each node (u, v, h) in layer 1 corresponds to a directed edge $(u, v) \in E$ in H, and has the same flow value $h_{(u,v)} = f_{(u,v)}$ as in H.
3. Each node (u, v, h) in any other layer $l \in \{2, \ldots, \ell\}$ has at most two predecessors in the previous layer $l - 1$. If (u, v, h) has two predecessors, then they have the form (u, w, g), (w, v, g') corresponding to a length-two path between u, v; this means that (u, v, h) is obtained by "merging" (a part of the) flows (u, w, g) and (w, v, g'). If (u, v, h) has only one predecessor, it must have the form (u, v, g). This means that (u, v, g) had some residual flow $h \leqslant g$ that was not merged with any other node.
4. The nodes at each layer $l \in \{1, \ldots, \ell - 1\}$ satisfy flow conservation with the succeeding layer $l + 1$, i.e. for each node (u, v, h) at any layer l having successors $\Pi(u, v, h)$ at layer $l + 1$, we have $\sum_{(x,y,g)\in\Pi(u,v,h)} g_{(x,y)} = h_{(u,v)}$.
5. The nodes at the top-most layer ℓ are only of the form (s, t, h), and have total flow value equal to $|f|$, i.e. these nodes all correspond to parallel (s, t) edges, which together account for all of the s-t flow f.

The size of this data-structure is measured in terms of the total number of nodes and edges it contains, where the edges denote successor-predecessor relationships between nodes across consecutive layers. If the size and the number of layers of a flow decomposition DAG are both low, we can efficiently perform various PRAMcomputations with low work and depth therein. Of particular importance to our approximate max-flow algorithm, for a η-size ℓ-layer flow decomposition DAG, we can compute the second vertex (which is a neighbor of s) and penultimate vertex (which is a neighbor of t) of every flow path in layer ℓ simultaneously with $O(\eta)$ work and $O(\ell + \log \eta)$ depth

using a simple algorithm; we show this in Lemma 5.5.5.

We now show that a small-size and low-depth flow decomposition DAG can indeed be found efficiently by slightly relaxing property 5 in Definition 5.5.1; specifically, given a parameter $\delta \in (0,1)$, we have that the nodes of the form $(s,t,\cdot)$ (i.e. parallel (s,t) edges) in layer ℓ together account for a $(1-\delta)$ fraction of the s-t flow value $|f|$ (at a small cost to its size and depth). This is sufficient for all our applications, since we are only concerned with *approximate* max flows. The precise guarantees are given in the following main theorem of this section.

Theorem 5.5.2 (Parallel Flow Decomposition)**.** *There exists a PRAMalgorithm* `flow-decomp` *that given any parameter $\delta \in (0,1)$ and a polynomially bounded s-t flow specified by a weighted, directed graph $H = (V, E, f)$ with flow value $|f|$, finds with high probability a flow decomposition DAG $\mathcal{D}$ of $\ell = O(\log(n/\delta))$ layers such that at the topmost layer ℓ, the total value of the flow captured by nodes $(s,t,\cdot)$ representing (parallel) (s,t) edges is at least $(1-\delta)|f|$. The algorithm* `flow-decomp` *has total work $O(m \ \text{polylog}(n)\log(n/\delta))$ and the total depth is $O(\text{polylog}(n)\log(n/\delta))$. The total number of nodes, and edges representing successor-predecessor relationships between nodes across consecutive layers in $\mathcal{D}$ are both bounded by $O(m\log(n/\delta))$.*

The idea behind our proof of Theorem 5.5.2 can be illustrated by the following relatively intuitive process[22]: repeatedly "shortcut" the flow graph H by replacing a length-two flow path $u \to w \to v$ with a single edge (u,v) having maximal flow value $h_{(u,v)} = \min\{g_{(u,w)}, g_{(w,v)}\}$, and a residual edge $(x,y) \in \{(u,w),(w,v)\}$ having flow value $h_{(x,y)} = |g_{(u,w)} - g_{(w,v)}|$ iff there is any non-zero leftover flow not accounted for by this shortcut edge (u,v). Consequently in the flow-decomposition DAG, the nodes $(u,w,g_{(u,w)})$ and $(w,v,g_{(w,v)})$ become the predecessors of this "shortcut edge" $(u,v,h_{(u,v)})$, and the node $(x,y,g_{(x,y)})$ becomes the predecessor of the residual edge $(x,y,h_{(x,y)})$ if there is any leftover flow. In order to achieve low depth, our objective is to find a collection of length-two flow paths that together account for a large fraction of the (total ℓ_1 norm of the) flow that does not directly go from s to t which we can then shortcut in parallel. We show that we can find such a collection of flow paths efficiently with the following technical lemma.

[22]This is only an illustration of our idea, see Section 5.5.3 and the proof of Proposition 5.5.4 for what we actually do.

Lemma 5.5.3. *Let f be an s-t flow of value $|f|$ specified by a weighted, directed multigraph $H = (V, E, f)$ containing source vertex $s \in V$, sink vertex $t \in V$ with no edges directly connecting s to t and with no self-loops, and the flow on any edge being given by $f : E \to \mathbb{R}^+$. Then we can find in $O(m \text{ polylog}(n))$ work and $O(\text{polylog}(n))$ depth, a collection of "shortcut paths" represented as tuples $\mathcal{P} := \left\{(e_i^{(1)}, e_i^{(2)}, h_i)\right\}$ where $e_i^{(1)} \neq e_i^{(2)} \in E$, $h_i \in \mathbb{R}^+$, along with a collection of "residual edges" given by $\mathcal{R} := \{(e_i, r_i)\}$ where $e_i \in E$ are disjoint, and $r_i \in \mathbb{R}^+$ such that*

1. *(Length-two paths) For each $i \in [|\mathcal{P}|]$, $e_i^{(1)}, e_i^{(2)}$ form a path of length two.*
2. *(Flow constraints) For each edge $e \in E$, the total flow accounted for by the shortcut paths involving this edge, which is given by the summation of h_i over the tuples in $\mathcal{P}$ in which e appears along with its residual capacity r_i if any (if e is present in $\mathcal{R}$), is exactly f_e:*

$$\sum_{i \in [|\mathcal{P}|]: e \in \left\{e_i^{(1)},\ e_i^{(2)}\right\}} h_i + \sum_{i \in [|\mathcal{R}|]: e = e_i} r_i = f_e.$$

3. *(Large ℓ_1-norm) With probability at least $1/15$, $\sum_{i \in [|\mathcal{P}|]} h_i \geqslant \|f\|_1 / 16$.*
4. *(Non-increasing flow-support) $|\mathcal{P}| + |\mathcal{R}| \leqslant |E|$.*

Proof. (Theorem 5.5.2) Given a flow f specified by a weighted directed graph $H = H_1$, we begin by creating the lowest layer 1 of our flow decomposition DAG $\mathcal{D}$ from H_1 as specified in property 2 of Definition 5.5.1. It is easy to see that this can be achieved with $O(m)$ total work and $O(1)$ depth. We also create a set $\mathcal{S}$, initially empty, to track tuples of the form $(s, t, h_{(s,t)})$ (i.e. direct (s, t) edges) which correspond to "fully processed" s-t flow paths. If H_1 contains such a direct edge, we delete it from H_1 and add the tuple $(s, t, f_{(s,t)})$ to $\mathcal{S}$. The subsequent proof (and algorithmic procedure) then follows from a repeated application of Lemma 5.5.3; the $l > 1$-th iteration, given as input an s-t flow specified by a weighted, directed *multigraph* H_{l-1}, we have:

1. Invoke Lemma 5.5.3 on H_{l-1} to find the desired shortcut paths $\mathcal{P}_l = \left\{(e_i^{(1)}, e_i^{(2)}, h_i)\right\}$, and residual edges $\mathcal{R}_l = \{(e_i, r_i)\}$. Set $\mathcal{S}_l$ to be initially empty.
2. Construct a new flow graph H_l, initially empty, as follows: for each tuple $(e_i^{(1)}, e_i^{(2)}, h_i) \in \mathcal{P}_l$,

add a directed edge (u_i, v_i) with flow value $f_{(u_i,v_i)} = h_i$, where u_i, v_i are the endpoints[23] of the length-two path $(e_i^{(1)}, e_i^{(2)})$. For each residual edge $(e_i, r_i) \in \mathcal{R}_l$, add a directed edge e_i with flow value r_i. Note that this may lead to the creation of parallel edges. Delete any edge (s, t) in H_l (may be multiple), as these correspond to "fully processed" flow paths that will be tracked separately in $\mathcal{S}_l$. Also delete any self-loops (edges with both endpoints being the same vertex) from H_l.

3. Construct layer l of the flow-decomposition DAG $\mathcal{D}$ as follows: for each tuple $\{e_i^{(1)}, e_i^{(2)}, h_i\} \in \mathcal{P}_l$, add to the l^{th} layer of $\mathcal{D}$ a node (u_i, v_i, h_i), where u_i, v_i are the endpoints[24] of the length-two path $(e_i^{(1)}, e_i^{(2)})$. We set the predecessors of (u_i, v_i, h_i) to be $(u_i^{(1)}, v_i^{(1)}, f_{(u_i^{(1)}, v_i^{(1)})})$ and $(u_i^{(2)}, v_i^{(2)}, f_{(u_i^{(2)}, v_i^{(2)})})$ from layer $l-1$, where $e_i^{(j)} = (u_i^{(j)}, v_i^{(j)})$ for $j \in \{1, 2\}$. For each residual edge $(e_i, r_i) \in \mathcal{R}_l$, add to the l^{th} layer of $\mathcal{D}$, a node (u_i, v_i, r_i) where $e_i = (u_i, v_i)$. We set the predecessor of (u_i, v_i, r_i) to be the node $(u_i, v_i, f_{(u_i,v_i)})$ from layer $l-1$. For each newly created node of the form (s, t, h) at layer l, add this node to $\mathcal{S}$. For each node $(s, t, h_i) \in \mathcal{S}_{l-1}$, add to the l^{th} layer of $\mathcal{D}$, a new node (s, t, h_i), and set its predecessor to be the corresponding (s, t, h_i) node from layer $l-1$. Finally, create $\mathcal{S}_l$ by adding all newly created $(s, t, \cdot)$ tuples in $\mathcal{S}$ to $\mathcal{S}_{l-1}$.

Observe that in each iteration, the updated flow specified by the multigraph H_l (prior to deleting (s, t) edges and self-loops if any) can trivially be routed in the preceding graph H_{l-1} by simply "undoing" the shortcutting that produced H_l. Moreover, after accounting for all the (s, t) edges specified by tuples in $\mathcal{S}_l$, the total flow value leaving s is preserved across all iterations in our procedure. Therefore, the "routability" property desired of the flow decomposition procedure is trivially satisfied by our aforementioned process.

We shall now prove the work and depth guarantees of this algorithm, along with the near linear-size and polylogarithmic depth of the resulting flow-decomposition DAG $\mathcal{D}$. Since in every iteration, the ℓ_1-norm of the flow f reduces by $\sum_{i \in \mathcal{P}} h_i$, which by property (3) of Lemma 5.5.3 is at least a constant-fraction of the ℓ_1-norm of the flow f at the start of the iteration with constant probability,

[23]Since we are only guaranteed an approximate max-flow f, it may contain circulations that we may discover in this process, i.e. the two endpoints u_i, v_i are identical. In this case, we can simply delete all the self-loops.

[24]If $u_i = v_i$, then skip this tuple without adding any nodes/edges.

we can deduce that after $\ell = \Theta(\log \frac{n\|f\|_1}{\delta|f|}) = \Theta(\log(n/\delta))$ iterations of the above process, with polynomially high probability (in n, follows by a straightforward Chernoff bound), the edges that remain in the flow graph H_ℓ (i.e. that do not directly go from s to t) contribute at most a δ fraction of the total (initial) amount of flow $|f|$ leaving s. Since in all our applications, we are only concerned with finding an approximate max-flow, we can safely ignore this residual flow for small enough δ. The second equality in the above bound follows by observing that $|f| \geqslant \min_{e\in E} w_e$, and $\|f\|_1 \leqslant \sum_{e\in E} w_e \leqslant m \cdot \max_{e\in E} w_e$, and the aspect ratio $\max_{e\in E} w_e / \min_{e\in E} w_e$ is assumed to be polynomially bounded.

In each iteration l, our procedure involves obtaining the relevant collection of tuples $\mathcal{P}_l, \mathcal{R}_l$ given an input flow multigraph H_{l-1}, which in turn are used to add a new layer l to the flow-decomposition DAG $\mathcal{D}$, and to update the flow multigraph $H_{l-1} \to H_l$. The total number of edges $|E_l|$ in the updated flow multigraph H_l is at most $|\mathcal{P}_l| + |\mathcal{R}_l| - |\mathcal{S}_l \setminus \mathcal{S}_{l-1}|$, which by property (4) of Lemma 5.5.3 is at most $|E_{l-1}|$, the total number of edges in the flow multigraph H_{l-1} from the previous iteration $l-1$. Moreover, the total number of nodes added to layer l in $\mathcal{D}$ is at most $|\mathcal{P}_l| + |\mathcal{R}_l| + |\mathcal{S}_{l-1}| = |\mathcal{P}_l| + |\mathcal{R}_l| - |\mathcal{S}_l \setminus \mathcal{S}_{l-1}| + |\mathcal{S}_l \setminus \mathcal{S}_{l-1}| + |\mathcal{S}_{l-1}| \geqslant |E_l| + |S_l|$. However, by property (4) of Lemma 5.5.3, we also have that $|\mathcal{P}_l| + |\mathcal{R}_l| \leqslant |E_{l-1}|$, due to which we have that $|\mathcal{P}_l| + |\mathcal{R}_l| + |\mathcal{S}_{l-1}| \leqslant |E_{l-1}| + |\mathcal{S}_{l-1}|$. Therefore, we can infer that the quantity $|E_{l'}| + |\mathcal{S}_{l'}|$ is non-increasing across l', due to which we can conclude that $|E_{l'}| + |\mathcal{S}_{l'}| \leqslant |E_1| + |\mathcal{S}_1| = |E|$. The above bounds, and the fact that the number of edges $|E|$ in the initial flow graph H provided as input to our algorithm is at most m combined with the computational guarantees of Lemma 5.5.3 gives us that any iteration of our algorithm (finding tuples, updating $H_{l-1} \to H_l$, creating a new layer in $\mathcal{D}$) can be implemented in $O(m \operatorname{polylog} n)$ work and $O(\operatorname{polylog} n)$ depth. Moreover, each new layer created in the flow decomposition DAG has at most m nodes, and at most $2m$ edges between it and the preceding layer (since every node has at most two predecessors from the previous layer). Since there are a total of $\Theta(\log \frac{n\|f\|_1}{\delta|f|}) = O(\log(n/\delta))$ iterations of our algorithm for polynomially bounded flows, we have that the total work and depth of our algorithm is $O(m \operatorname{polylog}(n) \log(1/\delta))$ and $O(\operatorname{polylog}(n) \log(1/\delta))$, respectively, and the size (number of nodes,edges) and depth of the flow decomposition DAG $\mathcal{D}$ produced is at most $O(m \log(n/\delta))$, and $O(\log(n/\delta))$, respectively as claimed. □

5.5.1. Proof of Lemma 5.5.3

One can formulate the problem as a (uncapacitated) b-matching problem. Specifically, we create a b-matching instance H_b from the flow multigraph $H = (V, E, f)$ (with no edge directly connecting s to t) as follows. For each edge $e \in E$, we create a vertex u_e with demand f_e. Then for every pair of edges $e^{(1)}, e^{(2)}$ that form a length-two path by sharing a vertex $v \in V$, we connect their corresponding vertices $u_{e^{(1)}}, u_{e^{(2)}}$ with an edge of infinite capacity. It suffices to find a large b-matching whose size is a constant fraction of $\|f\|_1$, the total ℓ_1-norm of the flow f.

First, we shall prove such a b-matching exists. To this end, let us consider a bipartite b-matching instance H'_b, constructed as follows. For each edge $e \in E$, we create two vertex copies $u_{e_{\text{in}}}$ and $u_{e_{\text{out}}}$ both with demand f_e. Then for every pair of edges $(e^{(1)}, e^{(2)})$ that form a length-two path by sharing a vertex $v \in V$, we add an edge between $u_{e^{(1)}_{\text{in}}}$ and $u_{e^{(2)}_{\text{out}}}$ with infinite capacity. We claim that there is a b-matching of size at least $\|f\|_1/2$, by the following simple construction. For each vertex $v \in V$ such that $v \neq s$ and $v \neq t$, we use a perfect b-matching between the vertices corresponding to the "in" copies of the incoming edges of v and the vertices corresponding to the "out" copies of the outgoing edges of v, whose existence is guaranteed by the flow conservation property. Then both vertices corresponding to every edge $e \in E$ is fully matched except for the vertices corresponding to the "out" copies of edges leaving the source s, and the vertices corresponding to the "in" copies of edges entering the sink t. By a simple charging argument that assigns the b-matching value for every vertex $v \in V$ to the "in" copies of the vertices corresponding to the incoming edges into v, we can bound the total demand of these aforementioned unmatched edges by at most $\|f\|_1/2$, which gives us our claim. Now consider for each edge $e \in E$, keeping either $u_{e_{\text{in}}}$ or $u_{e_{\text{out}}}$ uniformly at random and discarding the other vertex, letting the resulting subsampled graph be H_b. Then in expectation, observe that the maximum b-matching in the resulting subsampled graph is at least $\|f\|_1/8$, since each matched edge is kept with probability $1/4$ (i.e. if its two end points, one of which is a vertex corresponding to the "in" copy of an incoming edge, and the other is a vertex corresponding to the "out" copy of an outgoing edge are both sampled in the subsampled graph H_b). Therefore, by a standard Markov argument, we have that with probability at least $1/15$, the matching size in the subsampled graph is at least $\|f\|_1/16$.

We can in fact algorithmically compute such a matching efficiently in light of the above construction. As described above, we can subsample the vertices corresponding to the edge-copies in the graph H'_b and find the maximum b-matching in the resulting graph by locally finding, for each vertex $v \in V \setminus \{s,t\}$, the maximum b matching between the (subsampled) vertices corresponding to "in" copies of incoming edges into v, and (subsampled) vertices corresponding to "out" copies of outgoing edges from v in H. This is correct because after we keep either $u_{e_{\text{in}}}$ or $u_{e_{\text{out}}}$ and discard the other for each e in H, the graph becomes a union of disjoint connected components, where each component consists of the "in" copies of the incoming edges and the "out" copies of the outgoing edges of a single vertex. It then remains to show that we can efficiently find the maximum b-matching locally for every component, along with the non-increasing flow-support property for which it suffices to prove the following proposition.

Proposition 5.5.4. *Given two sets of elements $A = \{a_1, \ldots, a_{n_{\text{in}}}\}$ and $B = \{b_1, \ldots, b_{n_{\text{out}}}\}$, along with a positive function $f : A \cup B \to \mathbb{R}^+$ we can find in $O((n_{\text{in}}+n_{\text{out}})\ \text{polylog}(n_{\text{in}}+n_{\text{out}}))$ total work and $O(\log(n_{\text{in}}+n_{\text{out}}))$ depth, collections of tuples $\mathcal{P} := \{(x_i, y_i, h_i)\}$ where $x_i \in A, y_i \in B, h_i \in \mathbb{R}^+$, and $\mathcal{R} := \{(z_i, r_i)\}$ where $z_i \in A \cup B$ are disjoint and $r_i \in \mathbb{R}^+$ such that*

1. *(Flow constraints) For each $a \in A$ (and each $b \in B$), we have*

$$\sum_{i \in [|\mathcal{P}|]: x_i = a} h_i + \sum_{i \in [|\mathcal{R}|]: a = z_i} r_i = f_a, \quad \text{and}$$
$$\sum_{j \in [|\mathcal{P}|]: y_j = b} h_j + \sum_{j \in [|\mathcal{R}|]: b = z_j} r_j = f_b.$$

2. *(Non-increasing support) $|\mathcal{P}| + |\mathcal{R}| \leqslant n_{\text{in}} + n_{\text{out}}$*
3. *(Maximality) $\sum_{i \in [|\mathcal{P}|]} h_i = \min\left\{\sum_{a \in A} f_a, \sum_{b \in B} f_b\right\}$.*

Proof. The proof is standard and it appeared in earlier work like [ASZ20], though we describe it explicitly for completeness. We begin by computing the prefix sum of the values in both A and B; with some abuse of notation, let $A_k = \sum_{i=1}^{k} f_{a_i}$ and $B_k = \sum_{i=1}^{k} f_{b_i}$. By a standard PRAMalgorithm ([Akl97]), this procedure takes $O(n_{\text{in}}+n_{\text{out}})$ work and $O(\max\{\log n_{\text{in}}, \log n_{\text{out}}\}) = O(\log(n_{\text{in}}+n_{\text{out}}))$ depth. Next, we rank (sort) all the values in $\{A_i\}_{i \in [n_{\text{in}}]} \cup \{B_j\}_{j \in [n_{\text{out}}]}$, which takes

$O((n_{\text{in}}+n_{\text{out}})\log(n_{\text{in}}+n_{\text{out}}))$ work and $O(\log(n_{\text{in}}+n_{\text{out}}))$ depth; let C_k be the k-th ranked value in the resulting sorted prefix sum values, with c_k denoting the element a_i or b_j depending on whether $C_k = A_i$ or $C_k = B_j$. If there are ties (i.e. the total value A_i of the first i elements of A is exactly equal to the total value B_j of the first j elements of B for some i, j), then set $c_k = a_i$, and drop the $C_{k'}$ term corresponding to B_j after setting b_j to be the successor of c_k, which we define next. For each element c_k, find its successor $c_{\text{succ-}k}$ to be the element from the other array corresponding to the smallest prefix sum that is at least as large as C_k, i.e., if $c_k = a_i$ for some i (i.e. $C_k = A_i$), then the successor $c_{\text{succ-}k} = b_j$ where $j = \arg\min_{\ell: B_\ell \geqslant C_k = A_i} B_\ell$. If there is no such value (i.e. $C_k = A_i > B_{n_{\text{out}}}$), then we set the successor $c_{\text{succ-}k}$ to be a dummy element $\perp$. The case where $c_k = b_j$ follows symmetrically. We also let $C_0 = 0$ so that the subsequent process is well defined. Finally let C be the resulting processed, sorted array, and let n_{tot} be its length (i.e. array C contains $C_0, C_1, \ldots, C_{n_{\text{tot}}-1}$). This procedure can also be done in $O((n_{\text{in}}+n_{\text{out}})\cdot\log(n_{\text{in}}+n_{\text{out}}))$ work and $O(\log(n_{\text{in}}+n_{\text{out}}))$ depth by the standard doubling trick. Now starting with both $\mathcal{P}, \mathcal{R}$ being initially empty, do the following: for each $\ell \in \{1, \ldots, n_{\text{tot}}-1\}$ in parallel: if $c_\ell = a_i$, and $c_{\text{succ-}\ell} \neq \perp$, then add the tuple $(c_\ell, c_{\text{succ-}\ell}, C_\ell - C_{\ell-1})$ to $\mathcal{P}$; if $c_\ell = b_j$, and $c_{\text{succ-}\ell} \neq \perp$, then add the tuple $(c_{\text{succ-}\ell}, c_\ell, C_\ell - C_{\ell-1})$ to $\mathcal{P}$; otherwise, add the tuple $(c_\ell, C_\ell - C_{\ell-1})$ to $\mathcal{R}$. This entire process requires just $O(n_{\text{in}}+n_{\text{out}})$ work and $O(1)$ depth. This proves our computational guarantees.

We shall now prove the flow constraint, non-increasing support, and maximality properties outlined in the proposition statement. To prove the flow constraint property, consider any fixed element b_j for some j. By nature of our algorithm, b_j appears in tuples due to one of two reasons: either (a) it was the successor $b_j = c_{\text{succ-}k'}$ for some elements $c_{k'}$ (the number of such elements is $\geqslant 0$), or (b) when the element $C_k = B_j$ was processed by itself (exactly once if B_j was not tied with some A_i, in which case it was combined with its successor $c_{\text{succ-}k}$ to form a tuple and added to set $\mathcal{P}$ if $c_{\text{succ-}k} \neq \perp$, and to set $\mathcal{R}$ otherwise. If tied with some A_i, then this does not occur as there is no $k : C_k = B_j$). In the former, observe that b_j will be the successor of all elements $c_{k'} = a_i$ with value $C_{k'} = A_i$ where $B_{j-1} < C_{k'} = A_i \leqslant B_j$. If we sum the $h_{k'}$ values corresponding to all such k', including the final h_k value (which occurs only if there is a $k : C_k = B_j$), we get $\sum_{k': B_{j-1} < C_{k'} \leqslant B_j} h_{k'} + \mathbf{1}(\exists\, k : C_k = B_j) h_k = \sum_{k': B_{j-1} < C_{k'} \leqslant B_j} (C_{k'} - C_{k'-1}) + \mathbf{1}(\exists\, k : C_k = B_j)(C_k - C_{k-1}) = B_j - B_{j-1} = f_{b_j}$,

where the final equality follows by telescoping summation. The argument where $c_k = a_i$ follows symmetrically. The non increasing support property trivially follows by observing that the total number of tuples added to $\mathcal{P}, \mathcal{R}$ together is exactly $n_{\text{tot}} - 1 \leqslant n_{\text{in}} + n_{\text{out}}$ by definition of C. Lastly, to prove maximality, observe that the largest term $C_{n_{\text{tot}}-1}$ must be achieved at either $A_{n_{\text{in}}}$ or $B_{n_{\text{out}}}$. Let us assume that it is $B_{n_{\text{out}}}$, in which case it must be the case that every C_k term corresponding to some A_i value will have a successor in B, in which case the total capacity of $A_{n_{\text{in}}} = \sum_{i=1}^{n_{\text{in}}} f_{a_i}$ must be accounted for by the tuples in $\mathcal{P}$, since no a_i element will end up in the residual set $\mathcal{R}$. The argument for the other case where the largest term $C_{n_{\text{tot}}-1}$ is achieved at $A_{n_{\text{in}}}$ follows symmetrically. □

5.5.2. Computing Fractional Matching

It remains to show how, for each neighbor u of s, we may determine how much flow through u is routed through each neighbor of t. This allows us to use the flow to compute a (fractional) matching between the neighbors of s and the neighbors of t, which we use in the cut matching game (see Section C.2). Note also that in our use, the flow f that we want to decompose never contains any edges connecting s directly to t. The following lemma shows that this can be done using a flow-decomposition DAG; combined with Theorem 5.5.2, this lemma shows that this can be done in logarithmic depth and near-linear work.

Lemma 5.5.5. *Given a graph H, and vertices $s, t \in V(H)$, let S and T be the set of neighbors of s and t in H, respectively. Given a flow f without edges connecting s directly to t and a flow-decomposition DAG $\mathcal{D}$ of size η and depth ℓ, let P be the set of flow paths corresponding to the nodes of the form $(s, t, \cdot)$ at the final layer ℓ. There exists a $O(\ell + \log \eta)$ depth, $O(\eta)$ work PRAMalgorithm which computes $r_{x,y}$ for each $x \in S$ and $y \in T$ such that a total of $r_{x,y}$ units of flow is routed by the flow paths in P whose second vertex is x and penultimate vertex is y.*

Proof. For each node u of the form $(s, t, \cdot)$ at layer ℓ of the flow-decomposition DAG $\mathcal{D}$, our goal is to compute $\texttt{search}(u)$, which is the set $\{x, y\}$ such that $x \in S$ is the second node on the path represented by u and $y \in T$ is the penultimate node. To aid in presentation, for each DAG node v of the form $v = (s, a, \cdot)$ where $a \neq t$, we define $\texttt{search}(v)$ to be only the second node $x \in S$ along the path represented by v; we analogously define $\texttt{search}(v)$ to be only the penultimate node on the

path when $v = (a, t, \cdot)$. Note that we are only interested in $\texttt{search}(v)$ of nodes $u = (v_1, v_2, \cdot)$ of $\mathcal{D}$ such that $v_1 = s$ or $v_2 = t$.

Starting from each node u of the form $(s, t, \cdot)$ at level ℓ, in parallel we recursively compute $\texttt{search}(u)$ as follows:

- If u has exactly one predecessor p, return $\texttt{search}(p)$.
- If u is at level 1 in $\mathcal{D}$, then either $u = (s, x, \cdot)$ for some $x \in S$ or $u = (y, t, \cdot)$ for some $y \in T$ (recall that the initial given flow f does not contain edges directly connecting s to t); return x (or y, respectively).
- Otherwise, u has two predecessors u_1 and u_2.
  - If $u = (s, t, \cdot)$, then $u_1 = (s, a, \cdot)$ and $u_2 = (a, t, \cdot)$, for some $a \in V(H)$. Return $\{\texttt{search}(u_1), \texttt{search}(u_2)\}$.
  - Otherwise, $u = (v_1, v_2, \cdot)$, where either $v_1 = s$ or $v_2 = t$ (but not both), and $u_1 = (v_1, a, \cdot)$ and $u_2 = (a, v_2, \cdot)$ for some $a \in V(H)$. If $v_1 = s$, return $\texttt{search}(u_1)$, and if $v_2 = t$, return $\texttt{search}(u_2)$.

The correctness follows by induction on the level of u. If u is at level 1, $\texttt{search}(u)$ is trivially correct, and if u has exactly one predecessor p, then $\texttt{search}(u) = \texttt{search}(p)$, which is correct by induction. Otherwise, suppose u has two predecessors u_1 and u_2. If $u = (s, t, \cdot)$, then by the inductive hypothesis, $\texttt{search}(u_1)$ is the second node on the path represented by u_1 and $\texttt{search}(u_2)$ is the penultimate node on the path represented by u_2; since the path represented by u is the union of the paths represented by u_1 and u_2, it then follows that $\texttt{search}(u)$ is also correct. The correctness of $\texttt{search}(u)$ when $u = (v_1, v_2, \cdot)$, with $v_1 = s$ or $v_2 = t$, follows similarly.

The depth of the algorithm is bounded by the depth of $\mathcal{D}$, which is ℓ. Similarly, for work, the work to compute any *one* $\texttt{search}(u)$ is at most $O(\ell)$, as $\mathcal{D}$ has ℓ layers. So, the total work is $O(\zeta\ell) = O(\eta)$, where ζ is the number of nodes at level ℓ, and $\zeta\ell = O(\eta)$ by property 4 of Lemma 5.5.3. The desired values $r_{x,y}$ can then be computed by, for each u at level ℓ, adding the flow value of u to $r_{\texttt{search}(u)}$ (which is initially set to 0), which takes $O(\eta)$ work $O(\log \eta)$ depth by parallel summation. □

5.6. Applications

In this section, we discuss some notable applications of our parallel approximate max-flow algorithm. Namely, we show that our aforementioned result implies new or substantially improved parallel algorithms for (balanced) sparsest cuts, minimum-cost hierarchical clustering [Das16], fair-cuts [LNPS23] and approximate Gomory-Hu trees. The input instance to all these aforementioned problems are undirected, capacitated graphs $G = (V, E, c)$ with positive edge-weights that are assumed to be polynomially-bounded (or alternatively, a polynomially bounded ratio of maximum to minimum edge capacities).

5.6.1. Sparsest Cut and Balanced Min-cut

The sparsest cut problem is a classic problem in graph theory that informally asks to partition a given graph while removing as little edge-mass as possible. More precisely, given an undirected, weighted graph $G = (V, E, c)$ the objective is to find a cut $(X, V \setminus X)$ of minimum *sparsity*, which is formally defined as

$$\phi(X) = \frac{c(E(X, V \setminus X))}{\min\{|X|, |V \setminus X|\}},$$

where $c(E(X, V \setminus X))$ is the total weight of edges going across the cut $(X, V \setminus X)$. A closely related problem, the β-balanced minimum cut asks for a cut $(X, V \setminus X)$ such that the smaller side of the partition has at least βn vertices for a given parameter $\beta > 0$, and the total edge weight $c(E(X, V \setminus X))$ is minimized among all such partitions. The β-balanced sparsest cut is similarly defined as a cut whose i). smaller side has at least βn vertices and ii). sparsity is minimized among all such partitions.

All the aforementioned problems are known to be NP-hard, and the best-known polynomial-time algorithms achieve $O(\sqrt{\log n})$-approximation [ARV09b] to their corresponding objective, albeit at the expense of a large (sequential) polynomial running time. For the balance constrained variants of these cut problems, bicriteria approximations that allow multiplicative factors on both the balance parameter as well as the cut size are also commonly studied. This notion can be formally defined as follows.

Definition 5.6.1 (Bicriteria approximation for balanced cut problems)**.** Let $\beta' < \beta \leqslant \frac{1}{2}$ be real numbers, and let $(X^*, V \setminus X^*)$ be an optimal β-balanced min-cut (resp. β-balanced sparsest cut of G). We say that a cut $(X, V \setminus X)$ is an (α, β')-bicriteria approximation of the β-balanced min-cut (resp. β-balanced sparsest cut) if

1. $(X, V \setminus X)$ is a β'-balanced cut, and;
2. Approximation guarantees:
   (a) For β-balanced sparsest cut, $\phi(X) \leqslant \alpha \cdot \phi(X^*)$.
   (b) For β-balanced min-cut, $c(E(X, V \setminus X)) \leqslant \alpha \cdot c(E(X^*, V \setminus X^*))$.

While algorithms for sparsest and balanced-min cuts have been developed for the distributed CONGEST model of computation [KM15, CS19], no parallel PRAMalgorithms with nearly-linear work and polylogarithmic depth are known. We resolve this state-of-the-affair by designing the first PRAM$(\operatorname{polylog} n, \operatorname{polylog} n)$-bicriteria approximation for both problems using our parallel approximate max-flow algorithm. Formally,

Theorem 5.6.2. *There is a randomized PRAM algorithm that given an undirected weighted graph $G = (V, E, c)$, computes with high probability a cut $(X, V \setminus X)$ that achieves*

- *An $O(\log^3 n)$-approximation for sparsest cut;*
- *An $\left(O\left(\log^3 n\right), O\left(\frac{\beta}{\log^2 n}\right)\right)$-bicriteria approximation for β-balanced sparsest cut;*
- *An $\left(O\left(\frac{\log^3 n}{\beta}\right), O\left(\frac{\beta}{\log^2 n}\right)\right)$-bicriteria approximation for β-balanced min-cut.*

This algorithm has $O(m \cdot \operatorname{polylog} n)$ work and $O(\operatorname{polylog} n)$ depth.

Our algorithm builds upon the cut-matching game framework developed by [KRV06] that effectively reduces the computation of all of these aforementioned problems to polylogarithmically many single-commodity max-flow computations (as well as a flow-decomposition of the corresponding max-flow solutions). As a consequence, this framework provides algorithms for computing $\operatorname{polylog} n$ approximations to all these cut problems with work and depth that matching that of the max-flow (and flow-decomposition) oracle utilized in their implementations up to a $\operatorname{polylog} n$ factor. While the original idea of [KRV06] required an *exact* max-flow oracle in its implementation, [NS17] showed in a fairly straightforward extension that an approximate max-flow oracle also suffices to achieve

morally the same result.

Our algorithm, formally described in Algorithm 5.6.1, is a straightforward extension of this aforementioned result of [NS17] to the PRAMsetting. By crucially utilizing our parallel approximate max-flow result in Theorem 7, the parallel flow-decomposition result in Lemma 5.5.2, and the near-linear work, logarithmic depth flow-rounding algorithm of [Coh95], this algorithm achieves guarantees formally described in Lemma 5.6.3, which in turn imply the guarantees claimed in Theorem 5.6.2.

Lemma 5.6.3 (Parallel version of [NS17] Lemma B.18, cf. [KRV06])**.** *There is a randomized PRAM algorithm that given an undirected, weighted graph $G = (V, E, c)$, a sparsity parameter $\alpha > 0$, and a balance parameter $\beta = O(\frac{1}{\log^2 n})$, with high probability computes*

- *either an α-sparse cut X such that $\beta n \leqslant |X| \leqslant \frac{n}{2}$;*
- *or a graph H embeddable in G with congestion at most $O(\log^3 /\alpha)$ such that every $(\beta \log^2 n)$-balanced cut in H has sparsity at least $\Omega(1)$.*

The algorithm has $O(m \cdot \text{polylog}\, n)$ work and $O(\text{polylog}\, n)$ depth.

Proof. These aforementioned guarantees are achieved by Algorithm 5.6.1, and its correctness follows directly from [NS17] with no change to the proofs. Specifically the correctness for an α-approximate (unconstrained) sparsest cut follows from Lemma B.16 and Corollary B.21, and for an α-approximate, β-balanced sparsest cut follows from Lemma B.18 and Corollary B.22.

We now show that this algorithm admits a $O(m \cdot \text{polylog}\, n)$ work and $O(\text{polylog}\, n)$ depth PRAMimplementation. First, observe that in the t^{th} round of the cut-matching game, the cut player's strategy can implemented in $O(mt)$ work and $O(t \log n)$ depth as described in Lemma C.2.1. Now consider the matching player's strategy in the t^{th} round of the cut-matching game, which consists of $O(\log n)$ sub-rounds. We claim that each sub-round j of the matching player requires $O(m \cdot \text{polylog}\, n)$ work and $O(\text{polylog}\, n)$ depth. Observe that maintaining Sl and Sr and scaling edge weights can be done in $O(m)$ work and $O(1)$ depth. Since we pick $\varepsilon = O(1)$, the $(1 - \varepsilon)$ max-flow algorithm runs in $O(m \cdot \text{polylog}\, n)$ work and $O(\text{polylog}\, n)$ depth as in Corollary 7. Rounding the flow to an integral

Algorithm 5.6.1 `parallel-cut-matching-game`(G, α, β)

Input: Graph $G = (V, E, c)$, a sparsity parameter α, a balance parameter β.
Output:
Either Cut Case: a cut $(X, V \setminus X)$ such that $\phi(X) \leqslant \alpha$ and $\beta n \leqslant |X| \leqslant \frac{n}{2}$;
Or Expander Case: a graph H embeddable in G with congestion at most $O(\log^3 n/\alpha)$ such that every $(\beta \log^2 n)$-balanced cut in H has sparsity at least $\Omega(1)$.
Procedure:

for round $t = 1, \ldots, c_1 \log^2 n$, where c_1 is a sufficiently large constant **do**
  // The cut player:
  Sample a random n-dimensional unit vector $\boldsymbol{r}$ orthogonal to $\mathbf{1}$.
  if $t > 1$ **then**
    $\boldsymbol{u} \leftarrow \mathcal{M}_{t-1}(\mathcal{M}_t(\ldots(\mathcal{M}_1 \boldsymbol{r})))$, where for any $t' < t$, the $(n \times n)$ (sparse) matrix $\mathcal{M}_{t'}$ is the probability transition matrix corresponding to perfect matching output by the matching player in round t'.
  else
    $\boldsymbol{u} \leftarrow \boldsymbol{r}$
  Return cut $X_t \leftarrow$ vertices corresponding to the smallest $n/2$ entries in $\boldsymbol{u}$.
  // Matching player:
  Fix $\varepsilon = 1/10$, and let $c_2 = c_2(\varepsilon)$ be a sufficiently large constant.
  for sub-round $j = 1, \ldots, c_2 \log n$ **do**
    Maintain Sl_t^j and Sr_t^j: at the beginning, let $Sl_t^1 = X_t$; $Sl_t^j = Sl_t^j \setminus V(\mathcal{M}_t^{j-1})$, where $V(\mathcal{M}_t^{j-1})$ is the matching computed in the $(j-1)$-th sub-round of the matching player. Update Sr_t^j analogously.
    // Note that the sizes of Sl_t^j and Sr_t^j always remain equal by construction across all sub-rounds.
    Connect a source s to Sl_t^j and a sink t to Sr_t^j with edge capacities 1, and scale all other edge capacities in G by α^{-1}.
    Compute a $(1-\varepsilon)$-approximate max-flow, and round the flow to be integral using the algorithm in [Coh95]; also obtain the corresponding approximate min-cut $(C_t^j, V \setminus C_t^j)$.
    // by the dual variables of Sherman's framework.
    if $c(E(C_t^j, V \setminus C_t^j)) < |Sl_t^j| - \beta n$ **then**
      Output $(C_t^j, V \setminus C_t^j)$ as the desired α-sparse cut and terminate the game.
    else
      Compute a *partial* matching $\mathcal{M}_t^j$ between the vertex sets Sl_t^j, Sr_t^j given by the flow decomposition.
  Let $Sl_t^{J+1} = Sl_t^J \setminus V(\mathcal{M}_t^J)$, and $Sr_t^{J+1} = Sr_t^J \setminus V(\mathcal{M}_t^J)$, where $J = c_2 \log n$ is the final sub-round of the matching player.
  if $|Sl_t^{J+1}| = |Sr_t^{J+1}| < \beta n$ **then**
    Compute an arbitrary matching $\mathcal{M}_t'$ between Sl_t^{J+1}, Sr_t^{J+1}.
    Return the perfect matching $\mathcal{M}_t = \left(\bigcup \mathcal{M}_t^j\right) \cup \mathcal{M}_t'$ to be the union of the partial matchings across all sub-rounds of the matching player.
If the matching player outputs a cut in any (sub)round, then it is the desired α-sparse cut; otherwise, $H = \bigcup (\mathcal{M}_t \setminus \mathcal{M}_t')$ is the graph embeddable in G with low congestion.

solution can be done in $O(m)$ work and $O(\log n)$ depth by [Coh95]. Finally, since the edge capacities are bounded polynomials, we can use our parallel flow decomposition algorithm in Lemma 5.5.2 with $\delta = \frac{\varepsilon}{\|c\|_1}$. By Lemma 5.5.2, this takes $O(m \cdot \text{polylog}\, n)$ work and $O(\text{polylog}\, n)$ depth, and it preserves flow decomposition for a $(1-\varepsilon)$ approximate max-flow. Since there are at most $O(\log n)$ sub-rounds of the matching player in any (outer) round of our cut-matching game, the cut-player's strategy in the t^{th} round can also be implemented in $O(m\,\text{polylog}\, n)$ work and $O(\text{polylog}\, n)$ depth. Finally, since the number of rounds t of the cut-matching game is bounded by $O(\log^2 n)$, the total work and depth of our algorithm is bounded by $O(m\,\text{polylog}\, n)$ and $O(\text{polylog}\, n)$, respectively. □

The algorithm and analysis of Lemma 5.6.2. The algorithm follows from polylogarithmically-many parallel applications of Lemma 5.6.3. Concretely, when setting $\beta' = \beta/(10c\log^2 n)$ (or $\beta' = 0$ for the sparsest cut), we can guess the value of α as $(c_{\min}/n)\cdot 2^i$ for integer i and return the smallest guess value that returns a sparse cut. Any graph G whose eligible cuts have sparsity less than $\alpha/\log^3 n$ will not be embeddable for H, which gives us a cut whose sparsity is at least an $O(\log^3 n)$ approximation. Finally, note that by the balance of partition, an α-approximation of the β-balanced sparsest is always an $O(\alpha/\beta)$-approximation for the β-balanced min-cut.

To analyze the efficiency, under the assumption that the ratio between the maximum and the minimum capacities is polynomial, the geometrically-increasing guess can be done in parallel with an $O(\log n)$ multiplicative factor of work and $O(1)$ depth overhead. Each call to the algorithm of Lemma 5.6.3 takes $O(m \cdot \text{polylog}\, n)$ work and $O(\text{polylog}\, n)$ depth, and so the complete algorithm has total work $O(m \cdot \text{polylog}\, n)$ and depth $O(\text{polylog}\, n)$.

5.6.2. Minimum Cost Hierarchical Clustering

Before presenting our optimal sublinear time algorithms and lower bounds for the hierarchical clustering problem in CHAPTER 6, we first obtain near-optimal parallel algorithms for it as an application of our parallel maximum flow algorithm.

Hierarchical clustering is a fundamental data analysis tool used to organize data into a dendogram. Given data represented as an undirected weighted graph $G = (V, E, c)$, where the vertices represent datapoints and (positive) edge weights represent similarities between their corresponding end points,

the goal is to build a hierarchy, represented as a rooted tree $\mathcal{T}$; the leaves of this tree correspond to individual datapoints (vertices V), and the internal nodes correspond to a *cluster* $S \subseteq V$ consisting of their descendent leaves. Intuitively, this tree can be viewed as clustering the vertices of G at multiple levels of granularity simultaneously, with the clustering becoming increasingly fine-grained at deeper levels. [Das16] initiated the study of this problem from an *optimization* perspective by proposing the following cost function for similarity-based hierarchical clustering-

$$\mathsf{cost}_G(\mathcal{T}) = \sum_{(u,v)\in E} c_{uv} \cdot |\text{leaves}(\mathcal{T}_{uv})|, \tag{5.1}$$

where c_{uv} is the weight (similarity) of the edge $(u, v) \in E$ and $\mathcal{T}_{uv}$ is the subtree of $\mathcal{T}$ rooted at the least common ancestor of u and v, and $|\text{leaves}(\mathcal{T}_{uv})|$ is the number of descendent leaves in this subtree $\mathcal{T}_{uv}$. Intuitively, this objective imposes a large penalty for separating "similar" vertices at higher levels in the tree, thereby placing "similar" vertices closer together.

This minimization problem was shown to be NP-hard, and moreover, no polynomial time constant factor approximation factor was shown to be possible for this objective assuming the *small-set expansion* hypothesis [CC17]. Nevertheless, this objective is considered a reasonable one; [Das16] and a follow-up work by [CKMM18] show that it satisfies several properties desired of a "good" hierarchical clustering. As a consequence, this objective has been well studied in the literature (cf. [Das16, CC17, RP16, CNC18, CKMM18, ACL$^+$22, AKLP22b], and references therein), where it was shown to have strong algorithmic connections to the sparsest cut-problem. Specifically, [Das16, CC17, CKMM18] show that recursively partitioning the subgraph induced at each internal node using any α-approximate sparsest cut oracle results in a cluster tree that is a $O(\alpha)$ approximation for Dasgupta's objective.

While our results do imply a near-linear work, polylogarithmic depth parallel algorithm for computing polylog-approximate sparsest cuts, this by itself does not suffice to give a low-depth, work-efficient parallel algorithm for (approximately minimum-cost) hierarchical clustering. The reason is precisely that the sparsest cut subroutine may produce highly imbalanced partitions, resulting in a cluster tree with super-logarithmic depth. Due to the dependent nature of the recursively generated

subgraphs on which the sparsest cut oracle is invoked, the resulting clustering algorithm would have not just super-logarithmic depth, but also super-linear work.

Fortunately, the solution is relatively simple, which is to recursively partition the graph using balanced min-cuts (with $\beta = \Omega(1)$) instead, guaranteeing that the resulting hierarchy would have logarithmic depth. When utilizing such a balanced min-cut subroutine, [CC17, ACL$^+$22] show morally the same guarantees for the resultant hierarchical clustering as the unbalanced case. Specifically, they show that recursively partitioning the subgraph induced at each internal node using any (α, β')-bicriteria approximation oracle for β-balanced min-cuts ($0 < \beta' \leqslant \beta \leqslant 1/2$) results in a hierarchy that is an $O(\alpha/\beta')$ approximation for Dasgupta's cost function.

This result in combination with our parallel $O(\beta^{-1} \log^3 n, \beta \log^{-2} n)$-bicriteria approximation algorithm for β-balanced min-cuts presented in Theorem 5.6.2 directly gives us the first PRAMalgorithm for minimum-cost hierarchical clustering. Formally,

Theorem 5.6.4. *There is a randomized PRAM algorithm that given an undirected weighted graph $G = (V, E, c)$, computes with high probability, a hierarchical clustering tree $\mathcal{T}$ that is a $O(\log^5 n)$-approximation for Dasgupta's objective (Equation (5.1)). This algorithm has $O(m \cdot \operatorname{polylog} n)$ work and $O(\operatorname{polylog} n)$ depth.*

Combined with the simulation result of [KSV10, GSZ11b], the above result implies the first *fully-scalable* MPC algorithm for this problem; our algorithm computes a $O(\log^5 n)$-approximate minimum cost hierarchical clustering in $O(\operatorname{polylog} n)$ rounds, where each machine has local-memory $O(n^\delta)$ for any constant $\delta > 0$, and the total memory is $O(m \operatorname{polylog} n)$. Prior to this work, the state-of-the-art MPC algorithm for this problem required $\Omega(n \operatorname{polylog} n)$ local (per-machine) memory [AKLP22b].

5.6.3. Fair Cuts and Approximate Gomory-Hu Trees

In a recent work, [LNPS23] introduced the notion of fair cuts and showed that it is useful in several applications, one of them being the constructing approximate Gomory-Hu trees. Formally, a fair cut is defined as follows.

Definition 5.6.5 (Fair Cut [LNPS23])**.** Let $G = (V, E, c)$ be an undirected graph with edge ca-

pacities $c \in R^E_{>0}$. For any two vertices s and t, and parameter $\alpha \geq 1$, we say that a cut (S, T) is a α-fair (s, t)-cut if there exists a feasible (s, t)-flow such that $f(u, v) \geq \frac{1}{\alpha} \cdot c(e)$ for every $e \in E(S, T)$. Note that an α-fair (s, t)-cut is also an α-approximate (s, t)-min-cut but not vice-versa. We show that the results presented in our paper also extend to this stronger notion of approximate min-cuts, improving upon the algorithmic result of [LNPS23]. In particular, [LNPS23] give a PRAMalgorithm for computing a $(1 + \varepsilon)$-fair cut with $n^{o(1)}/\text{poly}(\varepsilon)$ depth and $m^{1+o(1)}/\text{poly}(\varepsilon)$ work. Importantly, the computational bottleneck in their approach is the construction and resultant quality of a congestion approximator for the input graph, in the sense that their algorithm has depth and work $\text{poly}(\alpha, \varepsilon^{-1}, \log n)$, and m $\text{poly}(\alpha, \varepsilon^{-1}, \log n)$, respectively, plus the depth and work required to construct such an α-congestion approximator. The same bottleneck persists in the application of their fair cut idea for computing isolating cuts and approximate Gomory-Hu trees. In their work, [LNPS23] utilize a *boundary-linked expander decomposition* of [CS19, GRST21] as their congestion approximator, which requires $n^{o(1)}$ depth and $m^{1+o(1)}$ work to construct, producing a $n^{o(1)}$-congestion approximator which gives them their PRAMresult[25]. Our new polylogarithmic depth, near-linear work construction of congestion approximators from Theorem 5.1.1 when used as a blackbox in the algorithm of [LNPS23] immediately imply the following improvements.

Theorem 5.6.6. *There is a randomized PRAM algorithm that given as input an undirected, weighted graph $G = (V, E, c)$, vertices $s, t \in V$, and desired precision $\varepsilon > 0$, with high probability, computes a $(1 + \varepsilon)$-fair (s, t)-cut in $O(m \text{ poly}(\varepsilon^{-1}, \log n))$ work and $O(\text{poly}(\varepsilon^{-1}, \log n))$ depth.*

Theorem 5.6.7. *There is a randomized PRAM algorithm that given as input an undirected, weighted graph $G = (V, E, c)$, and desired precision $\varepsilon > 0$, computes with high probability a $(1+\varepsilon)$-approximate Gomory-Hu tree in $O(m \text{ poly}(\varepsilon^{-1}, \log n))$ work and $O(\text{poly}(\varepsilon^{-1}, \log n))$ depth.*

[25] [LNPS23] only show their results for unweighted graphs since their parallel construction of congestion approximators are only for unweighted graphs; though they mention in their paper that they believe known techniques imply their same results for weighted graphs. We give an explicit construction even for the weighted case.

CHAPTER 6

OPTIMAL ALGORITHMS AND LOWER BOUNDS FOR SUBLINEAR TIME HIERARCHICAL CLUSTERING

Recall that in Section 5.6.2, we obtained near-optimal parallel algorithms for hierarchical clustering as an application of our parallel maximum flow algorithm.

In this chapter, we present our optimal algorithms and lower bounds for hierarchical clustering in the sublinear time setting. The results from this chapter are based on my work [AKLP22a] joint with Agarwal, Khanna, and Patil.

We start with an overview of our sublinear time algorithms in Section 6.1. We then give an overview of our sublinear time lower bounds in Section 6.2. Next, we present the key framework we develop for hierarchical clustering, which compresses the input similarity graph using a relaxed notion of cut sparsification, in Section 6.3. Leveraging this new framework, we then present our sublinear time algorithms in Section 6.4. We next present our sublinear time lower bounds in Section 6.5. Finally, we present our experimental results in Section 6.6.

6.1. Overview of Sublinear Time Algorithms

We begin by giving an overview of our sublinear time algorithmic result, which at its core is a new framework for hierarchical clustering which uses a relaxed notion of cut sparsification to compress the input graph.

6.1.1. Hierarchical Clustering by Relaxed Cut Sparsification

In their paper, [Das16] showed that the cost of a hierarchy $\mathcal{T}$, $\mathsf{cost}_G(\mathcal{T})$, can be viewed in two equivalent ways, the first being the one defined earlier in (1.2), and the other in terms of the *splits* induced by the internal nodes in the hierarchy: given a hierarchy $\mathcal{T}$ with each internal node corresponding to a binary split[26] where some subset of vertices $S \subseteq V$ of the input graph is

[26]This is without loss of generality since there always exists an optimal hierarchy that is binary.

partitioned into two pieces (S_ℓ, S_r), then

$$\mathsf{cost}_G(\mathcal{T}) := \sum_{\text{splits } S \to (S_\ell, S_r) \text{ in } \mathcal{T}} |S| \cdot w_G(S_\ell, S_r),$$

where for any disjoint subsets $S, T \subset V$, $w_G(S,T)$ is the total weight of the edges in G going between S and T. At this point, one might be tempted to think that if we could somehow construct a sparse representation of G such that the weights $w_G(S,T)$ are approximately preserved for any disjoint $S, T \subset V$, then the cost of every hierarchy would also be approximately preserved. Following this, we could run any desired offline algorithm on this representation with improved efficiency due to its sparsity without much loss in the quality of its solution. Unfortunately, this is just wishful thinking as such a representation can easily be shown to require $\Omega(m)$ time and space[27]. Our first contribution is to show there is in fact a *third equivalent view* of this same objective function in terms of global cuts in G, and the above alternate formulation serves as our starting point.

This result follows from two critical observations, the first of which is given any two disjoint $S, T \subset V$, we can compute $w_G(S,T)$ exactly as $w_G(S,T) = (1/2) \cdot (w_G(S, \overline{S}) + w_G(T, \overline{T}) - w_G(S \cup T, \overline{S \cup T}))$. We could stop here as the quantities on the right are all graph cuts, and it is well known [BK96] that one can construct a $\widetilde{O}(n)$ sized sparsifier that approximately preserves all graph cuts. Unfortunately, the distortion in $w_G(S,T)$ can be very large depending on the quantities on the right, and the cumulative error in $\mathsf{cost}_G(T)$ blows up with the depth of the tree which is even worse. Here is the second observation: the negative term $w_G(S \cup T, \overline{S \cup T})$ that internal node S contributes to the cost also appears as a positive term in its parent's contribution to the cost. We can pass this term as a *discount* in its parent's contribution to the cost, which after cascading gives a third view of (1.2).

$$\mathsf{cost}_G(\mathcal{T}) := \frac{1}{2} \cdot \left(\sum_{\text{splits } S \to (S_\ell, S_r) \text{ in } \mathcal{T}} \left(|S_r| \cdot w_G(S_\ell, \overline{S_\ell}) + |S_\ell| \cdot w_G(S_r, \overline{S_r}) \right) + \sum_{v \in V} w_G(\{v\}, \overline{\{v\}}) \right),$$

a linear combination of graph cuts. This gives a *strong blackbox reduction* to cut-sparsifiers; preserving graph cuts to a $(1 \pm \varepsilon)$ factor also preserves the cost of all hierarchies to a $(1 \pm \varepsilon)$ factor.

[27] Given such a sparsifier, by setting $S = \{u\}$ and $T = \{v\}$, one can recover whether or not edge (u, v) is present in G for any $u, v \in V$.

However, we are still far from done, as cut-sparsifiers cannot be computed efficiently in sublinear time. In fact, even to determine whether a graph is connected, one necessarily needs $\Omega(n^2)$ queries to the underlying graph [Gol17]. We therefore introduce a weaker notion of sparsification that, for any cut $(S, \overline{S})$, allows for an additive error of $\delta \min\{|S|, |\overline{S}|\}$ in addition to the usual multiplicative error of $(1 \pm \varepsilon)$. We term this generalization an (ε, δ)-cut sparsifier. A similar notion was also proposed in an earlier work by [Lee14], which unfortunately does not work here (see Section 6.4.1 for details). Our next result then shows that the distortion in the cost of any hierarchy under this weaker sparsifier is also bounded.

Theorem 6.1.1 (informal)**.** *Given any weighted graph G, an (ε, δ)-cut sparsifier of G preserves the cost of any tree $\mathcal{T}$ up to a multiplicative $(1 \pm \varepsilon)$ factor, and an additive $O(\delta n^2)$ factor.*

Therefore, if we could lower bound the cost of the optimal hierarchical clustering by some quantity C, we could set $\delta = \varepsilon C/n^2$. The above result would then imply morally the same result as that achieved by traditional cut-sparsifiers: preserving graph cuts in this ε, δ sense for a sufficiently small δ also preserves the cost of all hierarchies upto a $(1 \pm \varepsilon)$ factor. The last key result exactly establishes such a general purpose lower bound on the cost of any hierarchical clustering in a graph, which can be efficiently estimated in all models of computation we consider.

This chain of ideas results in the following new framework for sublinear HC given any parameter $\varepsilon > 0$: Compute the lower bound on the cost of an optimal HC which establishes the tolerable additive error in our (ε, δ) sparsifier, following which we efficiently compute the said sparsifier. We finally run any ϕ-approximate HC algorithm, which is guaranteed to find a $(1 + \varepsilon)\phi$-approximate HC tree. Our subsequent result give sublinear time construction of these (ε, δ)-cut sparsifiers in the query model.

6.1.2. Sublinear Time Algorithms in the Query Model

We consider the general graph model [Gol17] for sublinear time algorithms, where the input graph can be accessed via two[28] types of queries: (i) degree queries: given $u \in V$, returns degree d_u, and (ii) neighbour queries: given $u \in V$, $i \leqslant d_u$, returns the i-th neighbour of u. Note that this model

[28]This query model also allows a third type of queries: pair queries which answer whether an edge (u, v) exists or not. However, we do not need these queries in our algorithm.

can be easily implemented using an adjacency array representation of the graph (See Section 6.4 for a more detailed discussion). We first present the result for unweighted graphs, where it is easier to see the key intuition. Our main result in this model is a sublinear time construction of an (ε, δ)-sparsifier.

Theorem 6.1.2 (informal)**.** *There exists an algorithm that given query access to any unweighted graph G, and any parameters $\varepsilon, \delta \in (0, 1]$, can find an (ε, δ)-cut sparsifier of G w.h.p. in $\widetilde{O}(n/(\varepsilon^2\delta))$ time.*

Our algorithm is based on a simple yet elegant idea (which builds upon a slightly different idea proposed in [Lee14]; see Section 6.4.1 for a detailed discussion): if we embed a constant-degree expander with edge weights δ in an unweighted graph (with unit edge weights), then the effective resistance of every edge in the resulting composite graph is tightly bound in terms of the *effective* degrees of its incident vertices; the effective degree of a vertex is a weighted sum of its degree in the input graph and its degree in the expander. We can then leverage the effective resistance sampling scheme of [SS11b] to construct an $(\varepsilon, 0)$-cut sparsifier of this composite graph, which then is *deterministically* an (ε, δ)-cut sparsifier of the input graph with the sources of error being the usual multiplicative ε term due to sparsification itself, and the (small) additive δ term due to the (few) extra edges introduced by the expander. We can construct constant degree graphs that are expanders with high probability in sublinear time, and we show that there is an efficient rejection sampling scheme for sampling edges according to their effective resistances, giving the above result: an (ε, δ)-cut sparsifier with $\widetilde{O}(n/(\varepsilon^2\delta))$ edges in the same amount of time and queries. Moreover, the queries are completely *non-adaptive* assuming prior knowledge of vertex degrees. This construction of (ε, δ)-cut sparsifiers in conjunction with Theorem 6.1.1 then gives our sublinear time upper bound in this model.

Theorem 6.1.3 (informal)**.** *There exists an algorithm that given query access to any unweighted graph G with $m = \Theta(n^\zeta)$ for $\zeta \in [0, 2]$, can find a $(1 + o(1)) \cdot \phi$-approximate HC of G w.h.p. using $\widetilde{O}(g(n, \zeta))$ queries, where $g(n, \zeta) \leqslant n^{4/3}$ is given by $g(n, \zeta) = \max\{n, n^\zeta\}$ when $\zeta \in [0, 4/3]$, $g(n, \zeta) = \max\{n, n^{4-2\zeta}\}$ when $\zeta \in (4/3, 2]$. Moreover, given any (arbitrarily small) constant $\tau > 0$, the algorithm can find an $O(\sqrt{\log n})$-approximate HC of G w.h.p. in $\widetilde{O}(g(n, \zeta) + n^{1+\tau})$ time.*

It is interesting to observe that the query complexity $g(n, \zeta)$ reduces as the graph becomes denser. This is because the cost of the optimal HC increases with the density of the input graph, which allows us to tolerate a larger additive error in our cut sparsifier, thereby making it sparser. In Section 6.2 we also discuss lower bounds showing that this query complexity is also the best possible for any $\widetilde{O}(1)$-approximate algorithm. The sublinear time claim in Theorem 6.1.3 is implied by results from [She09] and [CKL$^+$22a] . Specifically, [She09] showed that, for any constant $\tau \in (0, 1/2)$, sparsest cuts and balanced separators can be approximated to within $O(\sqrt{\log n})$ in $\widetilde{O}(m)$ time plus $\widetilde{O}(n^\tau)$ maximum flow computations on graphs with $\widetilde{O}(n)$ edges, and [CKL$^+$22a] showed that a maximum flow on a graph of m edges can be computed in $O(m^{1+o(1)})$ time. These two combined with the fact that our sparsifier contains just $\widetilde{O}(g(n, \zeta))$ edges give us the desired running time bound.

We generalize the above result in Section 6.4.2 to weighted graphs by grouping edges according to geometrically increasing weights and constructing (ε, δ)-cut sparsifiers for each weight class, and get an algorithm with essentially the same worst case performance and $O(\log n)$ rounds of adaptivity.

6.2. Overview of Sublinear Time Lower Bounds

Note that the query complexity bound of our sublinear time algorithm is always at most $\widetilde{O}(n^{4/3})$, where the worst-case input is an unweighted graph with about $m \approx n^{4/3}$ edges. We note that our algorithm obtains an $O(\sqrt{\log n})$-approximation and (i) is completely *non-adaptive* on unweighted graphs assuming prior knowledge of vertex degrees, and has $O(\log n)$ rounds of adaptivity on weighted graphs; (ii) only uses degree queries and neighbor queries (no pair queries needed, see Footnote 28). We then show that $n^{4/3-o(1)}$ queries are indeed necessary for obtaining *any* $\widetilde{O}(1)$-*approximation* even in *unweighted* graphs and given *unlimited* adaptivity and access to *pair queries*.

Theorem 6.2.1 (informal)**.** *Let $\mathcal{A}$ be a randomized algorithm that, on any input unweighted graph with $\Theta(n^{4/3})$ edges, outputs with high probability a* polylog(n)*-approximate hierarchical clustering tree. Then $\mathcal{A}$ necessarily uses at least $n^{4/3-o(1)}$ queries.*

We briefly describe the family of hard graph instances that we use to prove this result. Roughly, a graph from such a family is generated by first taking a union of $n^{2/3}$ vertex-disjoint cliques of size $n^{1/3}$ each, and then connecting them by a random "perfect matching". More specifically, we

treat each clique as a *supernode*, and generate a perfect matching between these $n^{2/3}$ supernodes uniformly at random. Then if the i^{th} clique is matched to the j^{th} clique in the perfect matching, we will add about $n^{o(1)}$ edges between these two cliques, which are also chosen in a random manner. We show that, in order to output a good hierarchical clustering solution, it is necessary to discover a non-trivial portion of the edges that we add between the cliques, even though their number is relatively tiny compared to those within, and the latter task provably requires $n^{4/3-o(1)}$ queries.

While this plan looks intuitive, one has to be careful about not leaking information about the "perfect matching" between the cliques from the vertex degrees, which an algorithm knows a priori (or can otherwise acquire using $O(n)$ non-adaptive degree queries). In particular, once the inter-clique edges are added, one could tell that the vertices with degree higher than $n^{1/3} - 1$ are those participating in the perfect matching. Note that there are only $n^{2/3+o(1)}$ such vertices in total, and each of them has degree at most $O(n^{1/3})$. As a result, by probing all neighbors of these vertices, one can easily find all the inter-clique edges using $n^{1+o(1)}$ neighbor queries.

Our way around this issue is to also *delete* certain edges within the cliques based on what inter-clique edges we have added, so as to ensure that each vertex has the exact same degree of $n^{1/3} - 1$. This of course further complicates things as it increases the correlation between the edge slots — for instance, whenever an edge between a matched pair of cliques is revealed to the algorithm, missing edges within each clique are no longer independent. Consequently, our proof for this lower bound turns out to be considerably involved; we refer the reader to Section 6.5.2 for more details.

Theorem 6.2.1 shows that the worst-case query complexity of our algorithm is nearly optimal. Note that, however, for unweighted graphs, our algorithm also obtains improved query/time complexity when m is far from $n^{4/3}$. It is then natural to ask — are these improvements also the best possible? We answer this question in the affirmative. In particular, we show that one can push further the ideas we discussed above to get *a tight query lower bound for every graph density*. We summarize these lower bounds below. Note in particular that, for $m = \Theta(n^2)$, as we will show later, *any* hierarchical clustering achieves an $O(1)$-approximation, thus trivially 0 queries are sufficient.

Theorem 6.2.2 (informal)**.** *Let $\zeta \in [0, 2]$ be any constant. Let $\mathcal{A}$ be a randomized algorithm that, on any input unweighted graph with $\Theta(n^\zeta)$ edges, outputs with high probability a* polylog(n)*-approximate*

HC. Then $\mathcal{A}$ necessarily uses at least $\Omega(g(n,\zeta))$ queries, where $g(n,\zeta) = \max\{n, n^{\zeta - o(1)}\}$ when $\zeta \in [0, 4/3]$, $g(n,\zeta) = \max\{n, n^{4-2\zeta-o(1)}\}$ when $\zeta \in (4/3, 2)$, and $g(n,\zeta) = 0$ when $\zeta = 2$.

A formal version of this result is given in Section 6.5.

6.3. Hierarchical Clustering using (ε, δ)-Cut Sparsification

In this section, we shall present the key insight behind our algorithm result: *the hierarchical clustering cost function can equivalently be viewed as a linear combination of global cuts in the graph.* As a consequence, approximately preserving cuts in the graph also approximately preserves the cost of hierarchies in the graph, effectively reducing the hierarchical clustering problem to a cut-sparsification problem. However, there are hard lower bounds that refute an efficient sublinear time computation of traditional cut-sparsifiers in the query model. Therefore, we begin by introducing a weaker notion of cut sparsification, which we call (ε, δ)-cut sparsification.

Definition 6.3.1 ((ε, δ)-cut sparsifier)**.** Given a weighted graph $G = (V, E, w)$ and parameters $\varepsilon, \delta \geqslant 0$, we say that a weighted graph $\widetilde{G} = (V, \widetilde{E}, \widetilde{w})$ is an (ε, δ)-cut sparsifier of G if for all cuts $S \subset V$,

$$(1-\varepsilon) w_G(S) \leqslant w_{\widetilde{G}}(S) \leqslant (1+\varepsilon) w_G(S) + \delta \min\{|S|, |\overline{S}|\}$$

The above is a generalization of the usual notion of cut-sparsifiers (which are $(\varepsilon, 0)$-cut sparsifiers as per the above definition) that allows for an additive error in addition to the usual multiplicative error in any cut of the graph. A variant of this idea has been proposed before under the term *probabilistic (ε, δ)-spectral sparsifiers* in [Lee14] which was similarly motivated by designing sublinear time algorithms for (single) cut problems on unweighted graphs. However, as the name might suggest, the key difference between the prior work and ours is that the above bounds on the cut-values hold only in expectation (or any given constant probability) for any fixed cut in the former. Due to this limitation, we cannot use this previous work in a blackbox, and new ideas are needed.

We now show that for any two graphs that are close in this ε, δ sense, the cost of any hierarchy in these two graphs is also close as a function of these parameters, effectively allowing the use of (ε, δ)-cut sparsifiers in a *blackbox*.

Lemma 6.3.2. *Given any input weighted graph $G = (V, E, w)$ on n vertices, and an (ε, δ)-cut*

sparsifier $\widetilde{G}$ of G, then for any hierarchy $\mathcal{T}$ over the vertex set V, we have

$$(1-\varepsilon)\mathsf{cost}_G(\mathcal{T}) \leqslant \mathsf{cost}_{\widetilde{G}}(\mathcal{T}) \leqslant (1+\varepsilon)\mathsf{cost}_G(\mathcal{T}) + \frac{n(n+1)\delta}{2}.$$

Therefore, running a ϕ-approximate hierarchical clustering oracle $\mathcal{A}$ with input as the sparsifier $\widetilde{G}$ with $\varepsilon \leqslant 1/2$ produces a hierarchical clustering $\mathcal{T}_{\mathcal{A}}$ whose cost in G is at most

$$\mathsf{cost}_G(\mathcal{T}_{\mathcal{A}}) \leqslant (1+4\varepsilon)\phi \cdot \mathsf{cost}_G(\mathcal{T}^*) + n(n+1)\delta\phi,$$

where $\mathcal{T}^*$ is an optimal hierarchical clustering of G.

Proof. Consider any graph H (not necessarily G or $\widetilde{G}$) over vertex set V. Given any hierarchy $\mathcal{T}$ over the vertex set V, let S^0 be the root node with left and right children S^0_ℓ, S^0_r, respectively. Then we have the the cost of this hierarchy in H is given by

$$\begin{aligned}\mathrm{cost}_H(\mathcal{T}) &= \sum_{S\to(S_\ell,S_r)\in\mathcal{T}} |S|\cdot w_H(S_\ell,S_r)\\ &= |S^0|\cdot w_H(S^0_\ell,S^0_r) + \sum_{S\to(S_\ell,S_r)\in\mathcal{T}, S\neq S^0} |S|\cdot w_H(S_\ell,S_r).\end{aligned}$$

Now observe that since the split at the root S^0 is a partition of the entire vertex set V into S^0_ℓ, S^0_r, we have $w_H(S^0_\ell,S^0_r) = w_H(S^0_\ell) = w_H(S^0_r)$. Furthermore, observe that for any split of S, $S_\ell \cup S_r = S$, and therefore, we can represent the total weight of the edges crossing the split $w_H(S_\ell,S_r) = (\frac{1}{2})\cdot(w_H(S_\ell)+w_H(S_r)-w_H(S))$. Therefore,

$$\begin{aligned}\mathrm{cost}_H(\mathcal{T}) &= \frac{|S^0|}{2}(w_H(S^0_\ell)+w_H(S^0_r)) + \sum_{S\to(S_\ell,S_r)\in\mathcal{T}, S\neq S^0} \frac{|S|}{2}(w_H(S_\ell)+w_H(S_r)-w_H(S))\\ &= \sum_{S\to(S_\ell,S_r)\in\mathcal{T}} \left(\frac{|S|-|S_\ell|}{2}w_H(S_\ell) + \frac{|S|-|S_r|}{2}w_H(S_r)\right) + \sum_{v\in V}\frac{1}{2}w_H(v)\\ &= \frac{1}{2}\cdot\left(\sum_{S\to(S_\ell,S_r)\in\mathcal{T}} (|S_r|\cdot w_H(S_\ell)+|S_\ell|\cdot w_H(S_r)) + \sum_{v\in V} w_H(v)\right),\end{aligned}$$

Therefore, the hierarchical clustering cost function can equivalently be represented as a non-negative

weighted sum of cuts in a graph. We shall now use this reformulation of the clustering cost function to bound the error in the cost of any hierarchy $\mathcal{T}$ over a graph G and its (ε, δ)-sparsifier $\widetilde{G}$ as a function of the error in the cuts in these two graphs, which is parameterized by ε, δ. Our claimed lower bound is easy to see since for every cut $(S, \overline{S})$, $w_{\widetilde{G}}(S) \geqslant (1-\varepsilon) w_G(S)$, and therefore,

$$\mathsf{cost}_{\widetilde{G}}(\mathcal{T}) \geqslant \frac{(1-\varepsilon)}{2} \cdot \left(\sum_{S \to (S_\ell, S_r) \in \mathcal{T}} (|S_r| \cdot w_G(S_\ell) + |S_\ell| \cdot w_G(S_r)) + \sum_{v \in V} w_G(v) \right) = (1-\varepsilon)\mathsf{cost}_G(\mathcal{T}).$$

To show the upper bound, we have

$$\begin{aligned}
\mathsf{cost}_{\widetilde{G}}(\mathcal{T}) &\leqslant \frac{(1+\varepsilon)}{2} \cdot \left(\sum_{S \to (S_\ell, S_r) \in \mathcal{T}} (|S_r| \cdot w_G(S_\ell) + |S_\ell| \cdot w_G(S_r)) + \sum_{v \in V} w_G(v) \right) \\
&\quad + \frac{\delta}{2} \cdot \left(\sum_{S \to (S_\ell, S_r) \in \mathcal{T}} \left(|S_r| \cdot \min\{|S_\ell|, |\overline{S_\ell}|\} + |S_\ell| \cdot \min\{|S_r|, |\overline{S_r}|\} \right) + n \right) \\
&\leqslant (1+\varepsilon)\mathsf{cost}_G(\mathcal{T}) + \delta \cdot \left(\frac{n}{2} + \sum_{S \to (S_\ell, S_r) \in \mathcal{T}} |S_\ell| \cdot |S_r| \right).
\end{aligned}$$

Finally, we claim that for any binary hierarchical clustering tree $\mathcal{T}$ over n vertices (leaves),

$$\sum_{S \to (S_\ell, S_r) \in \mathcal{T}} |S_\ell| \cdot |S_r| \leqslant \frac{n^2}{2}$$

We shall prove this claim by induction on the number of leaves of $\mathcal{T}$. The base case is easy to see, which is a binary tree on 2 leaves. Assuming this claim holds for all binary trees on $n' < n$ leaves, consider any binary tree $\mathcal{T}$ with n leaves. Suppose the split at the root partitions the set of n leaves S^0 into sets S^0_ℓ and S^0_r. Let $\mathcal{T}_\ell, \mathcal{T}_r$ be the subtrees of $\mathcal{T}$ rooted at S^0_ℓ, S^0_r, respectively. Then we have

$$\sum_{S \to (S_\ell, S_r) \in \mathcal{T}} |S_\ell| \cdot |S_r| = |S^0_\ell| \cdot |S^0_r| + \sum_{S \to (S_\ell, S_r) \in \mathcal{T}_\ell} |S_\ell| \cdot |S_r| + \sum_{S \to (S_\ell, S_r) \in \mathcal{T}_r} |S_\ell| \cdot |S_r|.$$

Since both $|S_\ell^0|, |S_r^0| < n$, applying our induction hypothesis on the subtrees $\mathcal{T}_\ell, \mathcal{T}_r$ gives us that

$$\sum_{S \to (S_\ell, S_r) \in \mathcal{T}_\ell} |S_\ell| \cdot |S_r| \leqslant \frac{|S_\ell^0|^2}{2}, \text{ and } \sum_{S \to (S_\ell, S_r) \in \mathcal{T}_r} |S_\ell| \cdot |S_r| \leqslant \frac{|S_r^0|^2}{2}.$$

Substituting these bounds on the above sum proves our claim as

$$\sum_{S \to (S_\ell, S_r) \in \mathcal{T}} |S_\ell| \cdot |S_r| \leqslant |S_\ell^0| \cdot |S_r^0| + \frac{|S_\ell^0|^2 + |S_r^0|^2}{2} = \frac{|S_\ell^0 + S_r^0|^2}{2} = \frac{|S^0|^2}{2} = \frac{n^2}{2}.$$

Finally, observe that the ϕ-approximate hierarchical clustering oracle on input $\widetilde{G}$ finds a tree $\mathcal{T}_\mathcal{A}$ such that

$$\mathsf{cost}_{\widetilde{G}}(\mathcal{T}_\mathcal{A}) \leqslant \phi \cdot \mathsf{cost}_{\widetilde{G}}(\mathcal{T}), \qquad \forall \text{ hierarchies } \mathcal{T}. \tag{6.1}$$

Applying the above bound with $\mathcal{T} = \mathcal{T}^*$, an optimal hierarchical clustering of G gives us that

$$(1-\varepsilon)\mathsf{cost}_G(\mathcal{T}_\mathcal{A}) \overset{\text{Lem 6.3.2}}{\leqslant} \mathsf{cost}_{\widetilde{G}}(\mathcal{T}_\mathcal{A}) \overset{\text{Eq 6.1}}{\leqslant} \phi \cdot \mathsf{cost}_{\widetilde{G}}(\mathcal{T}^*) \overset{\text{Lem 6.3.2}}{\leqslant} (1+\varepsilon)\phi \cdot \mathsf{cost}_G(\mathcal{T}^*) + \frac{n(n+1)\delta\phi}{2}.$$

Therefore, for $\varepsilon \leqslant 1/2$, we have that

$$\mathsf{cost}_G(\mathcal{T}_\mathcal{A}) \leqslant \frac{1+\varepsilon}{1-\varepsilon}\phi \cdot \mathsf{cost}_G(\mathcal{T}^*) + \frac{n(n+1)}{2(1-\varepsilon)}\delta\phi \leqslant (1+4\varepsilon)\phi \cdot \mathsf{cost}_G(\mathcal{T}^*) + n(n+1)\delta\phi.$$

□

The above result shows that these weaker cut sparsifiers also approximately preserve the cost of any hierarchical clustering, but only up to an additive $O(\delta n^2)$ factor. Therefore, supposing we could efficiently estimate a lower bound OPT on the cost of an optimal hierarchical clustering in a graph G, we could then set the additive error $\delta = \varepsilon \mathrm{OPT}/n^2$, giving us that any ϕ-approximate hierarchical clustering for $\widetilde{G}$ is a $(1+5\varepsilon)\phi$-approximate hierarchical clustering for G. This implies that hierarchical clustering is effectively equivalent to efficiently computing an (ε, δ)-cut sparsifier with a sufficiently small additive error δ.

The following result fills in the final missing link in our chain of ideas by establishing a general-

purpose lower bound on the cost of any hierarchical clustering in an unweighted graph as a function of the number of vertices and edges in the graph.

Lemma 6.3.3. *Let G be any unweighted graph on n vertices and m edges. Then the cost of any hierarchical clustering in G is at least $4m^2/(3n)$.*

Proof. Given any unweighted graph $G = (V, E)$ over n vertices and m edges, fix any hierarchy $\mathcal{T}$ of the vertices V. In order to lower bound the cost of $\mathcal{T}$, we shall iteratively modify the "base graph" graph G by moving edges, strictly reducing the cost of $\mathcal{T}$ with each modification such that the final graph has a structure that makes the hierarchical clustering cost of $\mathcal{T}$ easy to analyze. In particular, the final graph would be such that each connected component is either a clique or two cliques connected together by some number of edges.

This is done as follows: given any hierarchy $\mathcal{T}$ of V, we perform a level order traversal over the internal nodes of $\mathcal{T}$, and at each node S, we modify the graph by pushing edges crossing the split (S_ℓ, S_r) down to lower level splits. Formally, let $S^1, \cdots, S^{n-1}$ be a level-order traversal over internal nodes of $\mathcal{T}$. We denote by $G^t = (V, E^t)$ the modified graph after visiting internal node S^t, with $G^0 = G$. Given G^t, we visit S^{t+1} and modify the graph as follows: if the subgraphs $G^t[S_\ell^{t+1}], G^t[S_\ell^{t+1}]$ induced by vertex sets S_ℓ^{t+1}, S_r^{t+1} respectively, are both cliques, then $G^{t+1} = G^t$; else move a maximal number of (arbitrary) edges crossing the split $(S_\ell^{t+1}, S_r^{t+1})$ to any (arbitrary) edge slots that are available in subgraphs $G^t[S_\ell^{t+1}], G^t[S_r^{t+1}]$ until either (a) the split $(S_\ell^{t+1}, S_r^{t+1})$ has no more edges going across in which case the two subgraphs become disconnected, or (b) both of the subgraphs become cliques with the edges remaining going across these cliques. We call the resulting graph G^{t+1}. Observe that the cost of $\mathcal{T}$ in G^{t+1} is at most the cost of $\mathcal{T}$ in G^t.

Let the final graph obtained after this traversal be G^{n-1}. It is easy to see that G^{n-1} is a collection of connected components, with each connected component being either a clique or two cliques with edges going across them, and that $\mathsf{cost}_{G^{n-1}}(\mathcal{T}) \leqslant \mathsf{cost}_G(\mathcal{T})$. In this graph G^{n-1}, (1) let $k_1, \ldots, k_r$ be the cliques, with k_j being the number of vertices in clique j, and (2) let $t_1, \ldots t_s$ be the connected components that are two cliques connecting by edges, where each $t_i = \{k_{i,1}, k_{i,2}, c_i\}$ with $k_{i,1}, k_{i,2}$ being the number of vertices in the two cliques of component t_i, and $c_i < k_{i,1} \cdot k_{i,2}$ being the number

of edges going across the two cliques. Then the cost of $\mathcal{T}$ on G^{n-1} is given by

$$\mathsf{cost}_{G^{n-1}}(\mathcal{T}) = \sum_{j=1}^{r} \frac{k_j^3 - k_j}{3} + \sum_{i=1}^{s} \left(\frac{k_{i,1}^3 - k_{i,1}}{3} + \frac{k_{i,2}^3 - k_{i,2}}{3} + (k_{i,1} + k_{i,2})c_i \right), \tag{6.2}$$

which follows by construction of G^{n-1} and Fact 2.7.2. We also observe that

$$\begin{aligned} n &= \sum_{j=1}^{r} k_j + \sum_{i=1}^{s} (k_{i,1} + k_{i,2}), \text{ and} \\ m &= \sum_{j=1}^{r} \binom{k_j}{2} + \sum_{i=1}^{s} \left(\binom{k_{i,1}}{2} + \binom{k_{i,2}}{2} + c_i \right). \end{aligned} \tag{6.3}$$

With these three observations, we shall now prove our claimed lower bound. We have that

$$
\begin{aligned}
m^2 &= \left[\sum_{j=1}^{r}\binom{k_j}{2} + \sum_{i=1}^{s}\left(\binom{k_{i,1}}{2} + \binom{k_{i,2}}{2} + c_i\right)\right]^2 \\
&= \Bigg[\sum_{j=1}^{r} k_j^{1/2}\left[k_j^{-1/2}\binom{k_j}{2}\right] + \\
&\quad \sum_{i=1}^{s} k_{i,1}^{1/2}\left[k_{i,1}^{-1/2}\left(\binom{k_{i,1}}{2} + \frac{c_i}{2}\right)\right] + \sum_{i=1}^{s} k_{i,2}^{1/2}\left[k_{i,2}^{-1/2}\left(\binom{k_{i,2}}{2} + \frac{c_i}{2}\right)\right]\Bigg]^2 \\
&\overset{(a)}{\leqslant} \left[\sum_{j=1}^{r} k_j + \sum_{i=1}^{s} k_{i,1} + \sum_{i=1}^{s} k_{i,2}\right] \\
&\quad \left[\sum_{j=1}^{r}\frac{1}{k_j}\binom{k_j}{2}^2 + \sum_{i=1}^{s}\frac{1}{k_{i,1}}\left(\binom{k_{i,1}}{2} + \frac{c_i}{2}\right)^2 + \sum_{i=1}^{s}\frac{1}{k_{i,2}}\left(\binom{k_{i,2}}{2} + \frac{c_i}{2}\right)^2\right] \\
&\overset{(b)}{=} \frac{n}{4}\Bigg[\sum_{j=1}^{r} k_j(k_j-1)^2 + \\
&\quad \sum_{i=1}^{s}\Big(k_{i,1}(k_{i,1}-1)^2 + k_{i,2}(k_{i,2}-1)^2 + 2c_i(k_{i,1}+k_{i,2}-2) + c_i^2(k_{i,1}^{-1}+k_{i,2}^{-1})\Big)\Bigg] \\
&\overset{(c)}{<} \frac{n}{4}\left[\sum_{j=1}^{r} k_j(k_j-1)^2 + \sum_{i=1}^{s}\left(k_{i,1}(k_{i,1}-1)^2 + k_{i,2}(k_{i,2}-1)^2 + 3c_i(k_{i,1}+k_{i,2})\right)\right] \\
&< \frac{3n}{4}\Bigg[\sum_{j=1}^{r}\frac{k_j(k_j-1)(k_j+1)}{3} + \\
&\quad \sum_{i=1}^{s}\left(\frac{k_{i,1}(k_{i,1}-1)(k_{i,1}+1)}{3} + \frac{k_{i,2}(k_{i,2}-1)(k_{i,2}+1)}{3} + c_i(k_{i,1}+k_{i,2})\right)\Bigg] \\
&\overset{(d)}{=} \frac{3n}{4}\mathsf{cost}_{G^{n-1}}(\mathcal{T}),
\end{aligned}
$$

where (a) follows by Cauchy-Schwarz inequality, (b) follows by (6.3), (c) follows by observing $c_i < k_{i,1} \cdot k_{i,2}$ due to which $c_i^2(k_{i,1}^{-1} + k_{i,2}^{-1}) < c_i(k_{i,1} + k_{i,2})$, and (d) follows from the cost of hierarchical clustering $\mathcal{T}$ in G^{n-1} established in (6.2). Therefore, we have that

$$\frac{4m^2}{3n} < \mathsf{cost}_{G^{n-1}}(\mathcal{T}) \leqslant \mathsf{cost}_{G}(\mathcal{T}),$$

for any hierarchical clustering $\mathcal{T}$ in any graph G on n vertices and m edges. □

6.4. Sublinear Time Algorithms

We now focus on the general graph (query) model [Gol17], where the input graph is accessible via the following two[29] queries: (a) Degree queries: given $v \in V$, returns the degree $d_G(v)$, and (b) Neighbour queries: given $v \in V$, $i \leqslant d_G(v)$, returns the i^{th} neighbour of v (neighbours are ordered arbitrarily). The objective is to compute a good hierarchical clustering of the input graph in time and queries sublinear in the number of edges in the graph. This problem becomes substantially more interesting in this setting, as finding an $(\varepsilon, 0)$-cut sparsifier necessarily takes linear $\Omega(n+m)$ queries. Therefore, the key to achieving such a result crucially depends upon being able to efficiently construct these weaker (ε, δ)-cut sparsifiers with a small additive error δ, which is the backbone of our sublinear time result. For simplicity, we begin by presenting our result for unweighted graphs, and then extend it to weighted graphs in subsection 6.4.2.

Theorem 6.4.1. *Given any unweighted graph $G = (V, E)$ with n vertices and $m = \alpha n^{4/3}$ edges accessible via queries in the general graph model, and any parameter $0 < \varepsilon \leqslant 1/2$, there exists an algorithm that*

(a) given a ϕ-approximate hierarchical clustering oracle, finds a $(1+\varepsilon)\phi$-approximate hierarchical clustering of G with high probability using $f(n, \alpha, \varepsilon)$ queries, and

(b) given any arbitrarily small parameter $0 < \tau < 1/2$, finds an $O(\sqrt{\tau^{-1} \log n})$-approximate hierarchical clustering of G with high probability using $\widetilde{O}(f(n, \alpha, \varepsilon) + n^{1+\tau})$ time and queries, where

$$f(n, \alpha, \varepsilon) = \begin{cases} O\left(\alpha n^{4/3}\right) & \alpha < 1 \\ \widetilde{O}\left(\varepsilon^{-3}(\alpha^{-2} n^{4/3} + n)\right) & \alpha \geqslant 1. \end{cases}$$

Note that unlike the sublinear space and communication settings, we cannot directly give a sublinear time $(1+\varepsilon)\phi$-approximation guarantee here; even though the rest of our algorithm (that

[29] As mentioned earlier, this model further allows for a third type of queries: (c) Pair queries: given $u, v \in V$, returns whether $(u, v) \in E$. This is equivalent to assuming the query oracle having internal access to both, an adjacency list representation (for degree and neighbour queries) as well as an adjacency matrix representation (for pair queries) of the input graph. However, our algorithm does not need pair queries, which further strengthens our algorithmic result in this model.

constructs the (ε, δ)-cut sparsifier) has a sublinear time and query complexity, the running time of the ϕ-approximate hierarchical clustering oracle to which we are given access can be arbitrarily large[30]. Therefore in this setting, we give a two-part result - the first is a *sublinear query*, $(1+\varepsilon)\phi$-approximation result, and the second is a *sublinear time and query*, $O(\sqrt{\log n})$-approximation result, which follows from a specific sublinear time implementation of a ϕ-approximate hierarchical clustering oracle with $\phi = O(\sqrt{\log n})$.

The query (and time) complexity in the above result is linear in the number of edges for sparse graphs with fewer than $n\sqrt[3]{n}$ edges, decays as $\widetilde{O}(\alpha^{-2}n^{4/3})$ for moderately dense graphs when the number of edges is in the range $n\sqrt[3]{n}$ and $n\sqrt{n}$, and is $\widetilde{O}(n)$ for dense graphs with more than $n\sqrt{n}$ edges. As we will see in our lower bounds, this complexity is essentially optimal for achieving a $\widetilde{O}(1)$-approximation in each of these three regimes.

Proof of Theorem 6.4.1. The proof of both parts of Theorem 6.4.1 relies on (ε, δ)-cut sparsifiers, which we show in Theorem 6.4.2, can be constructed with high probability in $\widetilde{O}(\varepsilon^{-2}\delta^{-1}n)$ time and queries. Assuming this construction, the sublinear query, $(1+\varepsilon)\phi$-approximation claim (Theorem 6.4.1 (a)) is relatively straightforward to see: we first determine the number of edges $m = \alpha n^{4/3}$ in the input graph by performing n degree queries. If the graph is sufficiently sparse ($m \leqslant n^{4/3}$), then we simply read the entire graph, which takes $O(m)$ neighbour queries. If not, then the lower bound established in Lemma 6.3.3 implies that the cost OPT of any hierarchical clustering in the input graph is at least $\alpha^2 n^{5/3}$. As a consequence, the additive error $\delta = \varepsilon \mathrm{OPT}/n^2 \geqslant \varepsilon\alpha^2 n^{-1/3}$ we can tolerate in our (ε, δ)-sparsifier is also relatively large. Such a sparsifier can then be constructed with high probability in $\widetilde{O}(\varepsilon^{-3}\max\{\alpha^{-2}n^{4/3}, n\})$ time and queries. The rest of the proof follows directly by Lemma 6.3.2, since the ϕ-approximate hierarchical clustering oracle uses only the (ε, δ)-cut sparsifier as input, and therefore, makes no additional queries to the input graph.

To prove the sublinear time, $O(\sqrt{\tau^{-1}\log n})$-approximation claim (Theorem 6.4.1 (b)) where $\tau \in (0, 1/2)$ is any arbitrarily small parameter, we complement the above proof with an instantiation of a sublinear time, ϕ-approximate hierarchical clustering oracle with $\phi = O(\sqrt{\tau^{-1}\log n})$. This

[30]Our sublinear query result more generally implies faster algorithms for hierarchical clustering without much loss in solution quality.

essentially reduces to a sublinear time, $O(\sqrt{\tau^{-1}\log n})$-approximation algorithm for balanced cuts (also called *balanced separators* in the literature). However, the algorithm of [ARV09a] cannot be used here due to its quadratic running time. Therefore, we instead refer to another result of [She09] that achieves $O(\sqrt{\tau^{-1}\log n})$-approximation for balanced separators by reducing this problem to $\widetilde{O}(n^{\tau})$ single-commodity max-flow computations for any given $\tau \in (0, 1/2)$. While [She09] could only achieve this in $\widetilde{O}(m + n^{3/2+\tau})$ time, the bottleneck being the $\widetilde{O}(m^{3/2})$ time flow-computation algorithm of [GR98] (with the sparsification result of [BK96] being used to improve this complexity to $\widetilde{O}(n^{3/2})$), we can leverage a very recent breakthrough [CKL$^+$22a] that gives an $\widetilde{O}(m^{1+o(1)})$ algorithm for exact single-commodity max-flows. This improves the running time of the algorithm of [She09] from $\widetilde{O}(m + n^{3/2+\tau})$ to $\widetilde{O}(m + n^{1+\tau})$ without any loss in the approximation factor. Since our (ε, δ)-cut sparsifier $\widetilde{G}$ is very sparse with $f(n, \alpha, \varepsilon)$ edges, we can find a $O(\sqrt{\tau^{-1}\log n})$-approximate balanced separator in $\widetilde{G}$ in sublinear $\widetilde{O}(f(n, \alpha, \varepsilon)+n^{1+\tau})$-time, for any given $\tau \in (0, 1/2)$. Although we use this subroutine repeatedly (at each split of the graph until we are left with singleton vertices), observe that at any level of the hierarchical clustering tree, the splits at that level together form a *disjoint partition* of $\widetilde{G}$. Now let the set of all internal nodes (splits) of the resultant hierarchical clustering tree at level $i \in [d]$ be $\mathcal{S}_i$, where d is the depth of the tree. Furthermore, for any internal node S in this tree, let m_S be the number of edges in the subgraph $\widetilde{G}[S]$ induced by the set of vertices S. Therefore, the running time of the recursive sparsest cut algorithm on $\widetilde{G}$ with the aforementioned $O(\sqrt{\tau^{-1}\log n})$-approximate oracle for balanced separators is given by

$$\widetilde{O}\left(\sum_{i\in[d]}\sum_{S\in\mathcal{S}_i}(m_S + |S|^{1+\tau})\right) \leqslant \widetilde{O}\left(\sum_{i\in[d]} f(n, \alpha, \varepsilon) + n^{1+\tau}\right).$$

Finally, observe that since the splits in the tree are balanced, i.e. a split $S \to (S_\ell, S_r)$ is such that $\min\{|S_\ell|, |S_r|\} \geqslant |S|/3$, the depth of this hierarchical clustering tree produced $d = O(\log n)$, which gives the total running time of the recursive sparsest cut algorithm on $\widetilde{G}$ as $\widetilde{O}(f(n, \alpha, \varepsilon) + n^{1+\tau})$, proving our sublinear time claim. □

We shall now present our sublinear time construction of (ε, δ)-cut sparsifiers for unweighted graphs.

6.4.1. A Sublinear Time (ε, δ)-Cut Sparsification Algorithm for Unweighted Graphs

Theorem 6.4.2. *There exists an algorithm that given a query access to an unweighted graph $G = (V, E)$ and parameters $0 < \delta \leqslant 1$, $0 < \varepsilon \leqslant 1/2$, can find a (ε, δ)-cut sparsifier of G with high probability in $\widetilde{O}(n\delta^{-1}\varepsilon^{-2})$ time and queries.*

Our (ε, δ)-cut sparsifier construction broadly builds upon ideas developed in [Lee14] for probabilistic spectral sparsifiers. At a high level, to achieve an additive error of δ, we embed a constant-degree expander $G_x = (V, E_x)$ with edge weights $\delta \leqslant 1$ on all n vertices in the input graph $G = (V, E)$. This trick gives a tight (and very friendly) bound on the effective resistance of every edge in the resultant composite graph $H = (V, E \cup E_x, w)$, which is a weighted graph with edge set consisting of the union of edges E in the input graph G, each having weight 1, and edges E_x in the constant-degree expander G_x, each with weight δ (edges in $E \cap E_x$ are assumed to be two parallel edges, one with weight 1 and the other with weight δ). This is useful as it allows for efficient sparsification of this composite graph using effective resistance sampling of [SS11b], with the sources of error being the usual multiplicative error due to sparsification itself, and a small additive error due to the few extra edges introduced by the expander. There are several well known $\widetilde{O}(n)$ time constructions of constant degree expanders, for example, a random d-regular graph is an expander with high probability [FKS89]. This roughly outlines the sparsification algorithm and proof of the above theorem.

A similar idea was used in [Lee14], with the key difference being that they embed an unweighted constant degree expander in a random δn subset of vertices. Since the set of vertices where the expander is embedded is random, it is easy to see why this gives a small additive error in expectation for any fixed cut, but could be very large for some cuts in the graph. Our construction on the other hand provides a sparsifier with stronger guarantees that hold for *every* cut. As outlined above, we start by showing that the effective resistance of any edge $(u, v) \in E \cup E_x$ is tightly bounded.

Lemma 6.4.3. *Given parameter $\delta \in (0, 1)$, and a composite graph $H = (V, E \cup E_x, w)$ where $G = (V, E)$ is an arbitrary input graph with edges of weight 1, and $G_x = (V, E_x)$ is a constant-degree*

expander graph with edges of weight δ, then for any edge $(u,v) \in E \cup E_x$, we have

$$\frac{1}{2}\left(\frac{1}{d_G(u)+\delta d_{G_x}(u)}+\frac{1}{d_E(v)+\delta d_{G_x}(v)}\right) \leqslant R_H(u,v) \leqslant O\left(\frac{\log n}{\delta}\left(\frac{1}{d_G(u)+\delta d_{G_x}(u)}+\frac{1}{d_E(v)+\delta d_{G_x}(v)}\right)\right),$$

where $R_H(u,v)$ is the effective resistance of edge (u,v) in graph H, and for any vertex $u \in V$, $d_G(u), d_{G_x}(u)$ are the degrees of vertex u in graphs G and G_x, respectively.

Proof. For any edge $(u,v) \in E \cup E_x$, let us assume without loss of generality that $k := d_H(u) \leqslant d_H(v)$. The proof of our upper bound on the effective resistance $R_H(u,v)$ relies on a basic property of expander graphs: there are $\Omega(k)$ edge-disjoint paths, each of length at most $O(\log n)$ connecting u to v. Since each edge on these paths has weight at least δ, by Rayleigh's monotonicity principle, the effective resistance between (u,v) can be no more than a graph containing exactly k edge-disjoint paths, each of length $O(\log n)$ with each edge on this path having resistance $1/\delta$, which gives us that

$$R_H(u,v) \leqslant O\left(\frac{\log n}{\delta k}\right) \leqslant O\left(\frac{\log n}{\delta}\left(\frac{1}{d_H(u)}+\frac{1}{d_H(v)}\right)\right) \leqslant O\left(\frac{\log n}{\delta}\left(\frac{1}{d_G(u)+\delta d_{G_x}(u)}+\frac{1}{d_G(v)+\delta d_{G_x}(v)}\right)\right)$$

To show this many edge-disjoint, short paths property of expanders, we consider two possibilities: either $k < n/\log n$, in which case let $\{u_i\}_{i=1}^k$ be the neighbours of u, and let $\{v_i\}_{i=1}^k$ be a set of k neighbours of v, chosen and ordered arbitrarily. Then the well known multicommodity flow result of [Fri01] already guarantees the existence of these short edge-disjoint paths connecting every u_i to v_i. In the case that $k \geqslant n/\log n$, consider the (unweighted) subgraph $H_x = (V, E_x \cup E_u \cup E_v)$ induced by the expander edges E_x and edges $E_u, E_v \subseteq E$ incident on vertices u, v in G, respectively. We first claim that the min u-v cut in H_x contains at least $k/2$ edges; let $(S, \overline{S})$ be the min u-v cut, with $|S| \leqslant n/2$ and $s \in \{u,v\}$ being the vertex in S. Furthermore, let $k_s \geqslant k$ be the number of neighbours of s, with $k'_s \leqslant k_s$ of them being contained in S. Now there are two possibilities, either *(a)* $k'_s < k_s/2$ in which case the cut $(S, \overline{S})$ already contains the $k_s - k'_s \geqslant k_s/2$ edges connecting

s to its remaining neighbours in $\overline{S}$, or (b) $k'_s \geqslant k_s/2$ in which case $(S, \overline{S})$ must necessarily cut at least $|S| \geqslant k_s/2$ edges in E_x due to expansion. Therefore, by the (integral) min-cut max-flow theorem, there are at least $k/2$ edge-disjoint paths from u to v. Moreover, we claim that at least half of these paths must be short, specifically, of length $O(\log n)$. To see this, observe that graph H_x contains just Cn edges for some constant C, which follows from that fact that u, v each can have at most n neighbours and G_x is a constant degree expander. Now let the integral flow which is of size $f \geqslant k/2 \geqslant n/(2\log n)$ be routed along arbitrary edge-disjoint paths $P_1, \ldots, P_f$. It is easy to see why at least $f/2$ of these paths must be of length at most $2C\log n$, because otherwise, the number of edges contained in just the long paths alone would exceed $(f/2)\cdot(2C\log n) > Cn$, the total number of edges in H_x which is a contradiction. Therefore, there are at least $k/4$ edge disjoint paths between u, v in $H_x \subseteq H$, each of length $O(\log n)$.

Now to prove the lower bound on the effective resistance of any edge $(u, v) \in E \cup E_x$, we add an extra vertex w and replace the edge (u, v) with two edges (u, w) and (w, v), each with weight/capacitance $2w_{uv}$ (doing this twice if edge $(u, v) \in E \cap E_x$, once for the edge (u, v) with $w_{uv} = 1$ and again for the edge with $w_{uv} = \delta$). We then apply Rayleigh's monotonicity principle by shorting all vertices other than u, v in the graph, which gives us that

$$\begin{aligned} R_H(u,v) &\geqslant \frac{1}{d_G(u) + \delta d_{G_x}(u) + w_{uv}} + \frac{1}{d_G(v) + \delta d_{G_x}(v) + w_{uv}} \\ &\geqslant \frac{1}{2}\left(\frac{1}{d_G(u) + \delta d_{G_x}(u)} + \frac{1}{d_G(v) + \delta d_{G_x}(v)}\right), \end{aligned}$$

where the final inequality follows from the fact that $w_{uv} < \min_{s\in\{u,v\}}\{d_G(s) + \delta d_{G_x}(s)\}$, which proves our claimed lower bound. □

This tight bound of the order $(d_G(u) + \delta d_{G_x}(u))^{-1} + (d_G(v) + \delta d_{G_x}(v))^{-1}$ on the effective resistances directly allows us to apply the effective resistance sampling scheme of [SS11b] outlined in Algorithm 6.4.1 to construct a spectral sparsifier of H, which is even stronger than the simple cut sparsifier we require. The following theorem then establishes the properties of the resulting sparsifier $\widetilde{G}$.

Algorithm 6.4.1 `sparsify`(G, p)

Input: Weighted graph $G = (V, E, w)$, edge sampling probabilities p such that $\sum_{e \in E} p_e = 1$, repetitions q.
Output: Sparsifier $\widetilde{G} = (V, \widetilde{E}, \tilde{w})$.
Procedure:

1: **for** $t = 1, \ldots, q$ **do**
2: Sample a random edge $e \in E$ according to p. Add e to $\widetilde{E}$ (if it does not already exist) and increase its weight $\tilde{w}_e$ by $w_e/(qp_e)$.

Theorem 6.4.4 (Theorem 1 + Corollary 6 of [SS11b])**.** *Given any weighted graph $G = (V, E, w)$ on n vertices with Laplacian L, let Z_e be numbers satisfying $Z_e \geqslant R_e/\alpha$ for some $\alpha \geqslant 1$ and $\sum_{e \in E} w_e Z_e \leqslant \sum_{e \in E} w_e R_e$. Then given any parameter $0 \leqslant \varepsilon \leqslant 1$, the subroutine* `Sparsify`$(G, p, q)$ *with sampling probabilities $p_e = w_e Z_e / (\sum_{e \in E} w_e Z_e)$ and $q = Cn \log n/\varepsilon^2$ for some sufficiently large constant C returns a graph $\widetilde{G}$ whose Laplacian $\widetilde{L}$, with high probability, satisfies*

$$\forall x \in \mathbb{R}^n \quad (1 - \varepsilon\sqrt{\alpha}) x^T L x \leqslant x^T \widetilde{L} x \leqslant (1 + \varepsilon\sqrt{\alpha}) x^T L x.$$

From the effective resistance bound established in Lemma 6.4.3, it is easy to see that sampling edges with parameter $Z_{uv} = (d_G(u) + \delta d_{G_x}(u))^{-1}/2 + (d_G(v) + \delta d_{G_x}(v))^{-1}/2$ satisfies the condition in Theorem 6.4.4 with $\alpha = O(\log n/\delta)$ for the graph $H = (V, E \cup E_x, w)$. Given this choice of parameters Z_e, it is easy to see that $\sum_{e \in E \cup E_x} w_e Z_e = n/2$, which gives us that the sampling probability for any edge $e \in E \cup E_x$ is given by

$$p_e = \frac{w_e}{n} \left(\frac{1}{d_G(u) + \delta d_{G_x}(u)} + \frac{1}{d_G(v) + \delta d_{G_x}(v)} \right), \tag{6.4}$$

for which there is a very simple sublinear time rejection sampling scheme given query access to G: sample a uniformly at random vertex $u \in V$, and toss a coin with bias $d_G(u)/(d_G(u) + \delta d_{G_x}(u))$ (degree query). If heads, sample a uniformly at random edge incident on u in G (neighbour query). Otherwise, sample a uniformly at random edge incident on u in G_x. The complete algorithm is given below.

It is easy to see that the above algorithm produces a graph $\widetilde{G}$ that has at most $O(n \log n/(\varepsilon')^2) =$

Algorithm 6.4.2 (ε, δ)-`sparsify`(G, ε, δ)

Input: Unweighted graph $G = (V, E)$ and $\varepsilon, \delta > 0$.
Output: Sparsifier $\widetilde{G}$.
Procedure:

1: Construct a constant degree expander $G_x = (V, E_x)$.
2: Let $H = (V, E \cup E_x, w)$ be the composite weighted graph with edge weights $w_e = 1$ for $e \in E$ and $w_e = \delta$ for $e \in E_x$.
3: Set $\varepsilon' = \varepsilon\sqrt{\delta/(C_1 \log n)}$ for a sufficiently large constant C_1, repetitions $q = C_2 n \log n/(\varepsilon')^2$ for a sufficiently large constant C_2.
4: Set sampling probabilities p, where for each edge $e \in E \cup E_x$, p_e is as defined in (6.4).
5: Sparsifier $\widetilde{G} =$ `sparsify`(H, p, q)

$O(n \log^2 n/(\delta\varepsilon^2))$ edges, and runs in time $O(n \log n/(\varepsilon')^2) = O(n \log^2 n/(\delta\varepsilon^2))$. We shall now prove that $\widetilde{G}$ is an (ε, δ)-sparsifier of G as claimed in Theorem 6.4.2.

Proof of Theorem 6.4.2. We start by observing that Theorem 6.4.4, by restricting to vectors $x \in \{0, 1\}^n$ (corresponding to partitions of V) and choice of $\varepsilon' = \varepsilon\sqrt{\delta/(C_1 \log n)}$ with a sufficiently large constant C_1, implies that with high probability, the sparsifier $\widetilde{G}$ produced by Algorithm 6.4.2 is such that

$$\forall S \subset V, \quad (1 - \varepsilon) w_H(S) \leqslant w_{\widetilde{G}}(S) \leqslant (1 + \varepsilon) w_H(S). \tag{6.5}$$

Now observe that for any cut $(S, \overline{S})$ in the composite graph H,

$$w_G(S) \leqslant w_H(S) = w_G(S) + w_{G_x}(S) \leqslant w_G(S) + \delta\Theta(\min\{|S|, |\overline{S}|\}),$$

where the final inequality follows by observing that the weight of any edge $e \in E_x$ is δ, and since the degree of any vertex in G_x is $\Theta(1)$, the number of edges in G_x that cross any cut $(S, \overline{S})$ is $\Theta(\min\{|S|, |\overline{S}|\})$. Combining the above bounds with (6.5) gives us the (ε, δ)-cut sparsification guarantees for $\widetilde{G}$ as

$$\forall S \subset V, \quad (1 - \varepsilon) w_G(S) \leqslant w_{\widetilde{G}}(S) \leqslant (1 + \varepsilon) w_G(S) + \Theta(\delta) \cdot \min\{|S|, |\overline{S}|\}.$$

□

6.4.2. Extension to Weighted Graphs

In this section, we extend our sublinear time results to weighted graphs $G = (V, E, w)$, where edges $e \in E$ take weights $1 \leqslant w_e \leqslant W$, where W is an upper bound on the maximum edge weight. Since there is no universally accepted query model for weighted graphs, we propose the following *generalization* where the algorithm can make (a) Degree queries: given $v \in V$, returns the degree $d_G(v)$, and (b) Neighbour queries: given $v \in V$, $i \leqslant d_G(v)$, returns both the i^{th} neighbour of v and the connecting edge weight, with the additional constraint that the neighbours are ordered by increasing edge weights (neighbours connected by equal weight edges are ordered arbitrarily). Note that this generalization reduces to the general graph model when all edge weights are equal. The following theorem describes our upper bound in this more general setting.

Theorem 6.4.5. *Let $G = (V, E, w)$ be any weighted graph with n vertices and edge weights taking values in a bounded range $[1, W]$. Given any parameter $0 < \varepsilon \leqslant 1/3$, let $m_i = \alpha_i n^{4/3}$ be the number of edges in G with weights in the interval $[(1+\varepsilon)^{i-1}, (1+\varepsilon)^i)$. Then given query access to G, there exists an algorithm that*

(a) given a ϕ-approximate hierarchical clustering oracle, finds a $(1+\varepsilon)\phi$-approximate hierarchical clustering of G with high probability using $\widetilde{O}\left((\varepsilon^{-1}\log W)\cdot(n + \max_i f(n, \alpha_i, \varepsilon))\right)$ queries, and

(b) given any arbitrarily small parameter $0 < \tau < 1/2$, finds an $O(\sqrt{\tau^{-1}\log n})$-approximate hierarchical clustering of G with high probability using at most time and number of queries $\widetilde{O}\left(n^{1+\tau} + (\varepsilon^{-1}\log W)\cdot(n + \max_i f(n, \alpha_i, \varepsilon))\right)$, where

$$f(n, \alpha, \varepsilon) = \begin{cases} O\left(\alpha n^{4/3}\right) & \alpha < 1 \\ \widetilde{O}\left(\varepsilon^{-3}(\alpha^{-2}n^{4/3} + n)\right) & \alpha \geqslant 1. \end{cases}$$

Before discussing this result, one might naturally ask whether this stricter requirement of ordering neighbours by weight is really necessary or whether it is possible to achieve a similar result for arbitrary or even random orderings. Towards the end of this section, we will show that this is unfortunately necessary; without the ordering constraint, no non-trivial approximation for hierarchical

clustering is possible unless a constant fraction of the edges in the graph are queried, and this holds even if we were to additionally allow pair queries: given $u, v \in V$, return whether $(u, v) \in E$ (and edge weight w_{uv} if affirmative).

At a high level, our sublinear time upper bound for weighted graphs is morally the same as that achieved in the unweighted case, with a $O(\varepsilon^{-1} \log W)$ hit to query and time complexity. Algorithmically, we build upon the ideas developed for the unweighted case. We begin by partitioning the edge set E of the input graph $G = (V, E, w)$ into weight classes, where the i^{th} weight class consists of all edges E_i with weights in the interval $[(1+\varepsilon)^{i-1}, (1+\varepsilon)^i)$. By construction, there are $\log_{(1+\varepsilon)} W \leqslant 2\varepsilon^{-1} \log W$ weight classes in total, with the edge sets $\{E_i\}_{i=1}^{\log_{(1+\varepsilon)} W}$ being a disjoint partition of E. We then approximately sparsify each *unweighted* subgraph $G'_i = (V, E_i)$ using our sublinear time (ε, δ)-Sparsify routine outlined in the previous section, and scale up all the edge weights of the resultant sparsifier $\widetilde{G}'_i$ by the maximum edge weight $W_i = (1+\varepsilon)^i$ of that class. Since for every weight class i, the weights of all the edges E_i in that class are within a $(1+\varepsilon)$ factor of each other, the resultant scaled sparsifier $\widetilde{G}_i$ is a good approximate sparsifier for the weighted subgraph $G_i = (V, E_i, w)$. Finally, since the input graph $G = (V, E, w)$ is partitioned into subgraphs $G_i = (V, E_i, w)$, the sum of the scaled sparsifiers $\widetilde{G}_i$ is a good sparsifier for the input graph. Given this sparsifier, the proof of (both claims of) Theorem 6.4.5 then follows identically as that of Theorem 6.4.1.

An important point to note here is that we do not need to explicitly construct the subgraphs G'_i corresponding to each of the weight classes $i \in [\log_{1+\varepsilon} W]$ (which would naively take $O(m)$ time) as our (ε, δ)-sparsification subroutine only requires query access to G'_i. This is easy to do in $\widetilde{O}(n)$ time for any weight class assuming the edges incident on vertices are sorted by weights. For any weight class i and any vertex v, the set of edges incident on v in subgraph G_i lie in the range of indices $[x_{i-1}(v), x_i(v) - 1]$ where for any weight class $j \in [\log_{1+\varepsilon} W]$, vertex $u \in V$, $x_j(u)$ is the first occurrence of an edge incident on u with weight at least $(1+\varepsilon)^j$. Both indices $x_i(v), x_{i-1}(v)$ can be found in $O(\log n)$ time and queries using binary search; the degree $d_{G'_i}(v) = x_i(v) - x_{i-1}(v)$, and the j^{th} neighbour of v in G'_i is simply the $(x_{i-1}(v) + j - 1)^{th}$ neighbour of v in G. Therefore, the total time and query complexity of setting up query access to G'_i is $O(n \log n)$. We now present a

formal proof of Theorem 6.4.5, which is achieved by Algorithm 6.4.3.

Algorithm 6.4.3 `weighted-sparsify`(G, ε)

Input: Weighted graph $G = (V, E, w)$, parameter $0 < \varepsilon \leqslant 1/3$.
Output: Sparsifier $\widetilde{G} = (V, \widetilde{E}, \widetilde{w})$.
Procedure:

For every vertex $v \in V$, $x_0(v) = 1$
for $i = 1, \ldots, \log_{(1+\varepsilon)} W$ **do**
    For every vertex $v \in V$, binary search for $x_i(v)$, the first occurrence of an edge incident on v with weight at least $(1+\varepsilon)^i$.
    Establish query access to $G'_i \leftarrow (V, E_i)$, where $E_i := \{e \in E : (1+\varepsilon)^{i-1} \leqslant w_e < (1+\varepsilon)^i\}$ using $\{(x_{i-1}(v), x_i(v))\}_{v \in V}$. Let $|E_i| = m_i = \alpha_i n^{4/3}$.
    if $\alpha_i \leqslant 1$ **then**
        Read $G_i = (V, E_i, w)$ entirely, and let this graph be $\widetilde{G}_i$.
    else
        Set additive error $\delta_i \leftarrow \varepsilon \cdot \min\{\alpha_i^2/n^{1/3}, 1\}$, and $W_i = (1+\varepsilon)^i$.
        $\widetilde{G}'_i \leftarrow (\varepsilon, \delta_i)$-`sparsify`$(G'_i)$, where $\widetilde{G}'_i = (V, \widetilde{E}_i, \widetilde{w}'_i)$
        Construct sparsifier $\widetilde{G}_i = (V, \widetilde{E}_i, \widetilde{w}_i = W_i \cdot \widetilde{w}'_i)$ with edge weights scaled by W_i.
$\widetilde{G} = \widetilde{G}_1 + \ldots + \widetilde{G}_{\log_{(1+\varepsilon)} W}$

Proof of Theorem 6.4.5. As with the analysis for unweighted graphs, we begin by establishing a lower bound on the cost of any hierarchical clustering for weighted graphs. Given any weighted graph $G = (V, E, w)$ and a parameter $0 < \varepsilon \leqslant 1/3$, we begin by partitioning the edge set into weight classes, where the i^{th} weight class consists of all edges E_i with weights in the interval $[(1+\varepsilon)^{i-1}, (1+\varepsilon)^i)$. Therefore, we have that the clustering cost of any hierarchy $\mathcal{T}$ on the weighted graph G is

$$\mathsf{cost}_G(\mathcal{T}) = \sum_{i=1}^{\log_{(1+\varepsilon)} W} \mathsf{cost}_{G_i}(\mathcal{T}) \geqslant \sum_{i=1}^{\log_{(1+\varepsilon)} W} (1+\varepsilon)^{i-1} \mathsf{cost}_{G'_i}(\mathcal{T}) \overset{\text{Lem 6.3.3}}{\geqslant} \sum_{i=1}^{\log_{(1+\varepsilon)} W} \frac{W_i \cdot |E_i|^2}{n}, \quad (6.6)$$

where the first inequality follows from the fact that the clustering cost function is monotone in edge weights, and every edge in $G_i = (V, E_i, w)$ has weight at least $(1+\varepsilon)^{i-1}$. We now claim that for every weight class i, the scaled sparsifier $\widetilde{G}_i$ is a $(O(\varepsilon), O(W_i \delta_i))$-sparsifier of the subgraph $G_i = (V, E_i, w)$. To see the lower bound, observe that for any cut $(S, \overline{S})$

$$w_{\widetilde{G}_i}(S) = W_i \cdot w_{\widetilde{G}'_i}(S) \overset{\text{Thm 6.4.2}}{\geqslant} W_i \cdot (1-\varepsilon) w_{G'_i}(S) \geqslant (1-\varepsilon) w_{G_i}(S), \quad (6.7)$$

where the final inequality follows from the fact that every edge in G_i has weight at most W_i. To see the upper bound, observe that for any cut $(S, \overline{S})$,

$$\begin{aligned} w_{\widetilde{G}_i}(S) = W_i \cdot w_{\widetilde{G}'_i}(S) &\overset{\text{Thm 6.4.2}}{\leqslant} W_i \cdot (1+\varepsilon) w_{G'_i}(S) + O(W_i \delta_i) \cdot \min\{|S|, |\overline{S}|\} \\ &\leqslant (1+\varepsilon)^2 w_{G_i}(S) + O(W_i \delta_i) \cdot \min\{|S|, |\overline{S}|\} \\ &\leqslant (1+3\varepsilon) w_{G_i}(S) + O(W_i \delta_i) \cdot \min\{|S|, |\overline{S}|\}, \end{aligned} \tag{6.8}$$

where the second inequality follows from the fact that every edge in G_i has weight at least $W_i/(1+\varepsilon)$. Since we have that the edge set $E = E_1 + \ldots + E_{\log_{(1+\varepsilon)} W}$, this directly gives us that the scaled sparsifier returned $\widetilde{G} = \widetilde{G}_1 + \ldots + \widetilde{G}_{\log_{(1+\varepsilon)} W}$ is a $(O(\varepsilon), O(\sum_i W_i \delta_i))$-cut sparsifier of G, where for any cut $(S, \overline{S})$,

$$(1-\varepsilon) w_G(S) \overset{\text{Eq. 6.7}}{\leqslant} w_{\widetilde{G}}(S) \overset{\text{Eq. 6.8}}{\leqslant} (1+3\varepsilon) w_G(S) + O\left(\sum_{i=1}^{\log_{(1+\varepsilon)} W} W_i \delta_i \right) \cdot \min\{|S|, |\overline{S}|\}. \tag{6.9}$$

By choice of each $\delta_i \leqslant \varepsilon |E_i|^2 / n^3$, we further have that

$$\sum_{i=1}^{\log_{(1+\varepsilon)} W} W_i \delta_i \leqslant \frac{\varepsilon}{n^2} \sum_{i=1}^{\log_{(1+\varepsilon)} W} W_i \cdot \frac{|E_i|^2}{n} \overset{\text{Eqn 6.6}}{\leqslant} \frac{\varepsilon}{n^2} \cdot \mathsf{cost}_G(\mathcal{T}), \qquad \forall \text{ hierarchies } \mathcal{T}.$$

Given this guarantee, the bound on the hierarchical clustering cost claimed in Theorem 6.4.5 (a) follows by a straightforward application of Lemma 6.3.2.

To complete this proof, the last thing we need to verify is the time and query complexity of Algorithm 6.4.3. We shall break down the complexity of this algorithm across the weight classes $i \in [\log_{1+\varepsilon} W]$. As described earlier, for any weight class i, establishing query access to the subgraph $G'_i = (V, E_i)$ requires at most $\widetilde{O}(n)$ time. Let $|E_i| = \alpha_i n^{4/3}$ be the number of edges in this subgraph. In the case $\alpha_i \leqslant 1$, this subgraph is sufficiently sparse and G_i is read entirely which takes $O(\alpha_i n^{4/3})$ time and queries. Otherwise $(\alpha_i > 1)$, in which case it is sparsified in $\widetilde{O}(\varepsilon^{-3} \max\{\alpha_i^{-2} n^{4/3}, n\})$ time and queries as established in Theorem 6.4.2. Therefore, the total complexity of processing a weight class i is $\widetilde{O}(n + f(n, \alpha_i, \varepsilon))$, where $f(n, \alpha, \varepsilon) = \widetilde{O}(\alpha n^{4/3})$ if $\alpha \leqslant 1$,

and $\widetilde{O}(\varepsilon^{-3}\max\{\alpha^{-2}n^{4/3}, n\})$ otherwise. Since there are $O(\varepsilon^{-1}\log W)$ weight classes in total, Algorithm 6.4.3 runs in time $\widetilde{O}(\varepsilon^{-1}n\log W + \sum_i f(n,\alpha_i,\varepsilon)) \leqslant \widetilde{O}((\varepsilon^{-1}\log W)\cdot(n + \max_i f(n,\alpha_i,\varepsilon))$. Lastly, for any given parameter $\tau \in (0, 1/2)$, the sublinear time, $O(\sqrt{\tau^{-1}\log n})$-approximation claim (Theorem 6.4.5 (b)) follows by the same ϕ-approximate hierarchical clustering oracle construction described in the proof of Theorem 6.4.1 combined with the fact that our (ε,δ)-cut sparsifier for the weighted graph G now contains $\widetilde{O}((\varepsilon^{-1}\log W)\cdot\max_i f(n,\alpha_i,\varepsilon))$ edges.

□

6.4.2.1 Necessity of Ordering Neighbours by Weight

We conclude this section by showing that the assumption that the adjacency list of each vertex u orders the neighbours of u by weight, is in fact necessary. Otherwise, no non-trivial approximation for hierarchical clustering is possible even when one is allowed to query a constant fraction of edges in the graph. We shall naturally consider only sufficiently dense graphs with $\Omega(n^{4/3})$ edges. While this isn't strictly necessary for our example, our upper (and lower) bounds allow us to simply read the entire graph otherwise, rendering the sparse regime uninteresting. While this is straightforward to see when the upper limit on edge weights $W = \text{poly}(n)$ is large, we can even show this for a relatively small $W = n^{1+\varepsilon}$ for any constant $\varepsilon > 0$. The example is as follows: consider an input graph $G = (V, E_1 \cup E_2, w)$ with n vertices, and an edge set of size m consisting of the union of two Erdős-Renyi random graphs, where $E_1 \sim \mathcal{G}_{n,p}$ for any $p > n^{-2/3}$ with all edges having weight 1 and $E_2 \sim \mathcal{G}_{n,1/3n}$ with all edges having weight $W = n^{1+\varepsilon}$ for some constant $\varepsilon > 0$. We can assume that edges in both E_1 and E_2 are given the larger weight.

We shall first establish an upper bound on the cost of the optimal hierarchical clustering in G, which we claim is at most $nm + O(nW\log n)$. To prove this, we shall use the fact that with probability at least $1 - 1/n$, (a) the subgraph $G_2 = (V, E_2)$ is a union of connected components, each either a tree or a unicyclic component, and (b) the degree of every vertex in G_2 is at most 3. The former is well known in the random graph literature, [ER$^+$60] and the latter follows from Bernstein's concentration inequality. Therefore, hierarchical clustering that first separates the different connected components of G_2, following which each connected component is partitioned recursively using a balanced sparsest

cut, i.e. the sparsest cut with a constant fraction of the remaining vertices on either side of the cut, will achieve a cost of at most $O(nW \log n)$. The remaining edges in E_1, regardless of how they are arranged can cumulatively add no more than $n|E_1|$ to the cost of this hierarchical clustering.

Now consider any (randomized) algorithm that performs at most $2m/9$ neighbour and pair queries in total, and let $\mathcal{T}$ be the hierarchical clustering returned by this algorithm. Consider a balanced cut $(S, \overline{S})$ in this tree, i.e. an internal node S with $\min\{|S|, |\overline{S}|\} \geqslant n/3$. Since the number of queries made is bounded by $2m/9$, there necessarily are at least $2n^2/9 - 2m/9 \geqslant n^2/9$ unqueried edge pairs from the cut $(S, \overline{S})$. Furthermore, there are at least $m - 2m/9 = 7m/9$ unqueried edges in G. For every unqueried edge, there is at least a constant $(n^2/9)/\binom{n}{2} \geqslant 2/9$ probability that it realized into an edge slot from the cut $(S, \bar{S})$, and then at least a $1/(3n)$ marginal probability that it came from E_2. Therefore in expectation, there are at least $(7m/9) \cdot (2/9) \cdot (1/3n) \geqslant m/(18n)$ edges from G_2, each having weight W that go across the cut $(S, \bar{S})$. Since $|S| \geqslant n/3$, the contribution of each heavy edge to the cost of $\mathcal{T}$ is at least $n/3 \cdot W$, and therefore, the expected cost of $\mathcal{T}$ is at least $(m/18n) \cdot (n/3) \cdot W = mW/54$ due to these heavy edges alone. Note that this argument holds even if the neighbours of every vertex are ordered randomly.

Now by comparing the cost of the optimal clustering, which is at most $nm + O(nW \log n)$, to the expected cost of the hierarchical clustering produced by an algorithm that makes at most $2m/9$ queries, which is $\Omega(mW)$, it is easy to see that the approximation ratio in expectation is $\Omega(n^{\varepsilon})$ when $W = n^{1+\varepsilon}$ and $m \geqslant n^{1+\varepsilon} \log n$.

6.5. Sublinear Time Lower Bounds

We note that, for unweighted graphs, our sublinear time algorithm requires only 2 rounds of adaptive queries, where the first round only needs to query vertex degrees. Thus if one assumes prior knowledge of vertex degrees, our algorithm is in fact *non-adaptive*. For weighted graphs, our algorithm requires at most $O(\log n)$ rounds of adaptive queries due to the binary searches. In any case, our algorithm makes at most $\widetilde{O}(n^{4/3})$ queries, where the worst-case input is an unweighted graph of about $\approx n^{4/3}$ edges.

We now show that, in a sharp contrast, even with unlimited adaptivity, our algorithm's query

complexity is essentially the best possible for any randomized algorithm that is able to compute a polylog(n)-approximate hierarchical clustering tree with high probability. In particular, we establish below tight query lower bounds when the input is an unweighted graph with $m = \Theta(n^\zeta)$ edges for any constant $\zeta \in [0, 2]$. By plugging in $\zeta = 4/3$ in **Case 4**, we get a matching lower bound for the worst-case input graph.

Case 1: $\zeta = 2$. Any binary hierarchical clustering tree has cost $O(n^3)$ (Fact 2.7.2), and by Lemma 6.3.3, the optimal cost is at least $\Omega(n^3)$. Thus trivially 0 queries are sufficient for $O(1)$-approximation.

Case 2: $\zeta \in [0, 1]$. It is not hard to show an $\Omega(n)$ query lower bound even for $o(n)$-approximation. Specifically, consider using a random matching of size $\Theta(n^\zeta)$ as a hard distribution, whose optimal hierarchical clustering cost is $\Theta(n^\zeta)$. However, any $o(n)$-query algorithm can only discover an $o(1)$-fraction of the matching edges, and with an $\Omega(1)$ fraction of the matching edges having high entropy, any balanced cut of the graph has nontrivial probability of cutting $\Omega(n^\zeta)$ matching edges, incurring a cost of $\Omega(n^{1+\zeta})$.

On the algorithmic side, one can simply probe all edges with $O(n)$ queries and then run any hierarchical clustering algorithm on the entire graph. Thus the query complexity for $\widetilde{O}(1)$-approximation is settled at $\Theta(n)$.

Case 3: $\zeta \in [3/2, 2)$. One can show an $\Omega(n)$ query lower bound for $\widetilde{O}(1)$-approximation, by considering an input graph obtained by randomly permuting the vertices of a union of vertex-disjoint cliques. We include a proof of this lower bound in Section 6.5.1.

On the algorithmic side, our sublinear time algorithm is able to obtain an $O(\sqrt{\log n})$-approximation using $\widetilde{O}(n)$ queries in this case, which is nearly optimal.

Case 4: $\zeta \in (1, 3/2)$. Let $\gamma := \zeta - 1 \in (0, 1/2)$. Our sublinear time algorithm obtains an $O(\sqrt{\log n})$-approximation using $\widetilde{O}(n^{\min\{1+\gamma, 2-2\gamma\}})$ queries. We show in Section 6.5.2 that this is nearly optimal even for $\widetilde{O}(1)$-approximation.

6.5.1. Lower bound for m between $n^{3/2}$ and n^2

Theorem 6.5.1 (Lower bound for m between $n^{3/2}$ and n^2)**.** *Let $\gamma \in [1/2, 1)$ be an arbitrary constant. Let $\mathcal{A}$ be a randomized algorithm that, on an input unweighted graph $G = (V, E)$ with $|V| = n$ and $|E| = \Theta(n^{1+\gamma})$, outputs a* polylog$(n)$*-approximate hierarchical clustering tree with probability $\Omega(1)$. Then $\mathcal{A}$ necessarily uses $\Omega(n)$ queries.*

We will show that there exists a distribution $\mathcal{D}$ over graphs with n vertices and $\Theta(n^{1+\gamma})$ edges, on which no deterministic algorithm using $o(n)$ queries can output a polylog(n)-approximate hierarchical clustering tree with probability $\geqslant .99$. This coupled with Yao's minimax principle [Yao77] will prove Theorem 6.5.1.

We define $\mathcal{D}$ such that a graph $G \sim \mathcal{D}$ is generated by first taking a union of $n^{1-\gamma}$ vertex-disjoint cliques of size n^γ, and then permuting the n vertices uniformly at random. More formally, we first pick a uniformly random permutation $\pi : [n] \to [n]$, and then let G be a union of vertex-disjoint cliques $C_1, \ldots, C_{n^{1-\gamma}}$ each of size n^γ such that C_i is supported on vertices

$$S_i := \{\pi((i-1)n^\gamma + 1), \ldots, \pi(in^\gamma)\}.$$

By Fact 2.7.2, we know that the optimal hierarchical clustering cost of each clique is $O(n^{3\gamma})$. Therefore, summing this cost over all cliques, we have:

Proposition 6.5.2. *The optimal hierarchical clustering tree of G has cost $O(n^{1+2\gamma})$.*

We now describe a process that interacts with any given deterministic algorithm $\mathcal{A}$ using $o(n)$ queries while generating a uniformly random permutation $\pi : [n] \to [n]$ along with its inverse function $\pi^{-1} : [n] \to [n]$. Specifically, we will generate π, π^{-1} by realizing them entry by entry adaptively based on the queries made be the algorithm. Thus, when realizing an entry of π or π^{-1}, we will always do so by conditioning on their already realized entries. Also note that since the degree of each vertex is the same (namely $n^\gamma - 1$), we will give the degree information to $\mathcal{A}$ for free at the start. The process then proceeds by the following two principles:

Principle 1: Upon a pair query between i, j, realize $\pi^{-1}(i), \pi^{-1}(j)$ and then answer the query

accordingly.

Principle 2: Upon a neighbor query about the ℓ^{th} neighbor of i, first realize $\pi^{-1}(i)$. Let k be such that the ℓ^{th} neighbor of i is $\pi(k)$. Then realize $\pi(k)$ and answer the query accordingly.

Clearly, each query triggers the realization of $O(1)$ entries of π and π^{-1}. Thus, after $\mathcal{A}$ terminates, the number of realized entries of π and π^{-1} is at most $o(n)$. Let $U \subset [n]$ with $|U| \geq (1-o(1))n$ be the set of indices whose π values are not realized, and similarly let $W \subset [n]$ with $|W| = |U| \geq (1-o(1))n$ be the set of indices whose π^{-1} values are not realized.

Let $\mathcal{T}$ be the hierarchical clustering tree output by $\mathcal{A}$, which we suppose for the sake of contradiction is polylog(n)-approximate. We first make $\mathcal{T}$ a full binary tree such that the bi-partition of each internal node is $[1/3, 2/3]$-balanced, during which we increase the cost of the tree by at most an $O(1)$ factor. We next consider the bi-partition of the root, which is a cut $(S, \bar{S})$ with $|S| \in [n/3, 2n/3]$.

Let $S' := S \cap W$ and $T' := \bar{S} \cap W$, and thus (S', T') is a bi-partition of W. Since $|W| \geqslant (1-o(1))n$, we have $|S'| \in [|W|/6, 5|W|/6]$. Since also $|U| \geqslant (1-o(1))n$, we have that for at least $\Omega(1)$ fraction of the cliques C_i's (which are supported on S_i's), we have

$$|\{(i-1)n^\gamma + 1, \ldots, in^\gamma\} \cap U| \geqslant n^\gamma/2.$$

For each such clique C_i, the number of edges within C_i that are across (S', T') is $\Omega(n^{2\gamma})$ with high probability. Therefore, the size of the cut $(S, \bar{S})$ is at least $\Omega(n^{1+\gamma})$ with high probability. This means that the cost of $\mathcal{T}$ is at least $\Omega(n^{2+\gamma})$, which together with $\gamma < 1$ contradicts $\mathcal{T}$ being polylog(n)-approximate.

6.5.2. Lower bound for m between n and $n^{3/2}$

Theorem 6.5.3 (Lower bound for m between n and $n^{3/2}$)**.** *Let $\gamma \in (0, 1/2)$ be an arbitrary constant. Let $\mathcal{A}$ be a randomized algorithm that, on an input unweighted graph $G = (V, E)$ with $|V| = n$ and $|E| = \Theta(n^{1+\gamma})$, outputs with $\Omega(1)$ probability a* polylog(n)*-approximate hierarchical clustering tree. Then $\mathcal{A}$ necessarily uses at least $n^{\min\{1+\gamma, 2-2\gamma\}-o(1)}$ queries.*

By Yao's minimax principle [Yao77], to prove Theorem 6.5.3, it suffices to exhibit a hard input distribution on which every *deterministic* algorithm using a small number of queries fails with nontrivial probability. Specifically, we will show that there exists a distribution $\mathcal{D}$ over graphs with n vertices and $\Theta(n^{1+\gamma})$ edges such that, on an input graph drawn from $\mathcal{D}$, any deterministic algorithm using $n^{\min\{1+\gamma,2-2\gamma\}-\delta}$ queries for any constant $\delta > 0$ can only output a polylog(n)-approximate hierarchical clustering tree with $o(1)$ probability.

The hard distribution. We start by defining the hard distribution $\mathcal{D}$ over graphs with n vertices and $\Theta(n^{1+\gamma})$ edges. Roughly speaking, we will generate an input graph G by first taking the union of a certain number of cliques $C_1, \ldots, C_k$ of equal size n/k, and then adding some artificially structured edges between them. We will then show that even the edges between the cliques are relatively tiny compared to those within, it is necessary to discover them in order to output a good hierarchical clustering solution.

More specifically, we will decide what edges to add between cliques based on the structure of a randomly generated "meta graph" H on k supernodes, with supernode i in H representing the clique C_i. We generate the meta graph H by picking a uniformly random perfect matching between the k supernodes (assuming for simplicity k is even). Then for each matched pair of supernodes i, j in the meta graph H, we will add between C_i and C_j a random bipartite matching of certain size (note that this matching is in the actual graph G rather than the meta graph H). Moreover, when adding the latter matching edges in G, we will also delete some edges inside C_i, C_j to ensure that every vertex has the exact same degree, so that an algorithm cannot tell which vertices participate in the meta graph's perfect matching by only looking at the vertex degrees. We will then show:

1. Any deterministic algorithm using $n^{\min\{1+\gamma,2-2\gamma\}-\delta}$ queries for any $\delta > 0$ can only discover an $o(1)$ fraction of the matching edges in the meta graph H.
2. If $\Omega(1)$ fraction of the matching edges have high entropy, an algorithm cannot output a polylog(n)-approximate hierarchical clustering tree with $\Omega(1)$ probability.

We now formally describe how we generate a graph G from $\mathcal{D}$. Let the vertices of G be numbered

1 through n. We divide the vertices into $n^{1-\gamma}$ groups $S_1, \ldots, S_{n^{1-\gamma}}$ each of size n^γ, where

$$S_i := \{(i-1)n^\gamma + 1, \ldots, in^\gamma\}.$$

We then generate the edges of G by the process in Figure 6.1.

1. Generate a meta graph H on supernodes numbered $1, \ldots, n^{1-\gamma}$ by picking a uniformly random perfect matching (of size $n^{1-\gamma}/2$) between them.
2. Initially, add a clique C_i of size n^γ to each vertex group S_i, and insert the clique edges into the adjacency list of G in an arbitrary order.
3. Let $t \leftarrow n^{\max\{0,3\gamma-1\}+\frac{1}{\sqrt{\log n}}}$. In what follows, we will add a matching of size $2t$ between each matched clique pair.
4. For each matched pair of supernodes i, j in the meta graph H:
   (a) Add a uniformly random *bipartite* matching $M_{i,j}$ of size $2t$ between S_i and S_j, and let $T_{i,j}$ denote the vertices matched by $M_{i,j}$ (thus $|T_{i,j} \cap S_i| = |T_{i,j} \cap S_j| = 2t$).
   (b) Inside S_i (resp. S_j), pick a uniformly random perfect matching of size t between vertices $T_{i,j} \cap S_i$ (resp. $T_{i,j} \cap S_j$), and delete its edges from clique C_i (resp. C_j).
   (c) Modify the adjacency list of the vertices in G by replacing the edges deleted at Step 4b with the edges added at Step 4a. This modification is valid because the degree of each vertex is preserved.

Figure 6.1: Generation of $G \sim \mathcal{D}$.

Proposition 6.5.4. *All vertices in G have degree exactly $n^\gamma - 1$.*

Proposition 6.5.5. *The optimal hierarchical clustering tree of G has cost $O(n^{1+2\gamma})$.*

Proof. We will construct a hierarchical clustering tree as follows. At the first level, we divide the entire vertex set into $n^{1-\gamma}/2$ clusters where each cluster is a connected component. This step incurs zero cost. We then construct a binary hierarchical clustering tree of each cluster arbitrarily. Since each cluster has $2n^\gamma$ vertices, the hierarchical clustering tree we construct for it has cost bounded by $O(n^{3\gamma})$ (Fact 2.7.2). Summing this upper bound over all $n^{1-\gamma}/2$ clusters finishes the proof. □

Analysis of deterministic algorithms on $\mathcal{D}$. Let $\mathcal{A}$ be a deterministic algorithm that makes $n^{\min\{1+\gamma,2-2\gamma\}-\delta}$ queries for some constant $\delta > 0$. Since all vertices have the same degree $n^\gamma - 1$ in G, we will give the degree information to A for free at the start. We shall then describe a process that interacts with the algorithm $\mathcal{A}$ while generating a $G \sim \mathcal{D}$. To that end, we first define the notion of revealed vertex groups.

Definition 6.5.6 (Revealed vertex groups)**.** At any given point of the algorithm, we say a vertex group S_i is *revealed by* $\mathcal{A}$ if at least one of the following is true:

Condition 1: At least $\frac{n^{2\gamma}}{10000t}$ pair queries involving vertices in S_i are made by $\mathcal{A}$.

Condition 2: At least $\frac{n^{2\gamma}}{10000t}$ neighbor queries on vertices in S_i are made by $\mathcal{A}$.

Condition 3: A pair query by $\mathcal{A}$ finds a pair $u, v \in S_i$ not connected by an edge.

Condition 4: A pair query or a neighbor query by $\mathcal{A}$ finds a pair $u \in S_i, w \notin S_i$ connected by an edge.

We now describe a process that answers queries made by $\mathcal{A}$ while *adaptively* realizing the edge slots and the adjacency list of G, as well as the perfect matching in the meta graph H. Whenever realizing a part, we will always do so following the distribution $\mathcal{D}$ *conditioned on* the already realized parts. This means that if a part is already realized or determined by other realized parts, realizing it again will not change it. The process proceeds according to the following three principles:

Principle 1: Upon a pair query, realize the corresponding edge slot and answer accordingly.

Principle 2: Upon a neighbor query, realize the corresponding entry of the adjacency list and answer accordingly.

Principle 3: As soon as a group S_i becomes revealed after a query, due to either large query count or what we have answered by **Principle 1** and **Principle 2**, right away do:

- Realize the supernode j that is matched to i in the meta graph H.
- Realize *all* edge slots incident on (and hence also all neighbors of) vertices in S_i, S_j.

At any given point of this process, we say a vertex group S_i is *realized* if all edge slots incident on S_i are realized. That is, the realized vertex groups are *exactly* those revealed by $\mathcal{A}$ and the ones matched to them. This in particular implies that a perfect matching has been realized between the realized vertex groups in the meta graph H, while none of the unrealized vertex group is matched. As a result, one can show that the queries made so far that involve unrealized vertex groups must have *deterministic* answers:

Proposition 6.5.7. *At any point of the algorithm $\mathcal{A}$, for the queries already made, we have:*

- *Every pair query between an unrealized vertex group and a realized one discovered no edge.*
- *Every pair query between two unrealized vertex groups discovered no edge.*
- *Every pair query within a same unrealized vertex group discovered an edge.*
- *Every neighbor query on a vertex in an unrealized vertex group found a neighbor within the same group.*

In what follows, we will consider the conditional distribution of $\mathcal{D}$ on all edge slots incident on realized vertex groups, which we denote by $\mathcal{D}_{\text{rz}}$. Note that $G' \sim \mathcal{D}_{\text{rz}}$ is *not* necessarily consistent with the answers we gave to the queries that involve unrealized vertex groups, though these answers are themselves deterministic by Proposition 6.5.7. By definition, a graph $G' \sim \mathcal{D}_{\text{rz}}$ can be generated by the process in Figure 6.2.

1. Add the edges incident on the realized vertex groups to G'.
2. Add the perfect matching between the realized vertex groups to the meta graph H.
3. Add a clique C_i of size n^γ to each unrealized vertex group S_i.
4. For each unrealized vertex group S_i:
   (a) If supernode i is not matched in the meta graph H, then match i to another uniformly random unmatched j, and change the edges within $S_i \cup S_j$ using Steps 4a-4c in Figure 6.1.

Figure 6.2: Generation of $G' \sim \mathcal{D}_{\text{rz}}$.

Proposition 6.5.8. *Consider generating $G' \sim \mathcal{D}_{\text{rz}}$ conditioned on that an unrealized S_i is matched to another unrealized S_j in the meta graph H. Then $G'[S_i \cup S_j]$ is consistent with previous answers with probability at least .998.*

Proof. First note that, when changing the edges within $S_i \cup S_j$ at Step 4a in Figure 6.2, the edges we delete from C_i (resp. C_j) distribute as a uniformly random matching of size t in C_i (resp. C_j), and the edges we add between S_i, S_j distribute as a uniformly random bipartite matching of size $2t$ between S_i, S_j, though these distributions are correlated.

Then note that $G'[S_i \cup S_j]$ is *not* consistent with previous answers only if (i) the slot of an edge we delete within S_i or S_j was queried by $\mathcal{A}$, or (ii) an edge we add between S_i, S_j was queried by $\mathcal{A}$. Since S_i, S_j are both unrevealed, they do *not* satisfy **Condition 1** or **Condition 2**. As a result,

we can bound the probability of $G'[S_i \cup S_j]$ being inconsistent with previous answers via a union bound by

$$2 \cdot \frac{2n^{2\gamma}}{10000t} \cdot \frac{t}{\binom{n^\gamma}{2}} + \frac{2n^{2\gamma}}{10000t} \cdot \frac{2t}{n^{2\gamma}} \leqslant .002,$$

which proves the proposition. □

We show that the number of realized vertex groups can be at most a $o(1)$ fraction of the total.

Proposition 6.5.9. *Upon termination of the algorithm $\mathcal{A}$, the total number of realized vertex groups S_i's is bounded by $o(n^{1-\gamma})$ with probability at least $1 - 1/n^4$.*

Proof. The number of vertex groups that satisfy **Condition 1** or **Condition 2** can be at most

$$\begin{aligned}
\frac{2 \cdot \#\text{queries}}{\frac{n^{2\gamma}}{10000t}} =& \frac{2n^{\min\{1+\gamma, 2-2\gamma\}-\delta}}{\frac{n^{2\gamma}}{10000t}} \\
=& \frac{20000 n^{\max\{0,3\gamma-1\}+\frac{1}{\sqrt{\log n}}} n^{\min\{1+\gamma,2-2\gamma\}-\delta}}{n^{2\gamma}} \qquad \text{(plugging in the value of } t\text{)} \\
=& 20000 n^{1-\gamma+\frac{1}{\sqrt{\log n}}-\delta} \\
\leqslant& n^{1-\gamma-\Omega(1)} \leqslant o(n^{1-\gamma}).
\end{aligned}$$

This means that the total number of realized vertex groups that satisfy **Condition 1** or **Condition 2** and those matched to them is at most $o(n^{1-\gamma})$.

We then bound the number of realized vertex groups that do not satisfy **Condition 1** or **Condition 2** and are not matched to those who satisfy **Condition 1** or **Condition 2**. Each such vertex group must be (matched to) a revealed one that satisfies **Condition 3** or **Condition 4**. We thus consider the probability that a query makes an unrealized vertex group satisfy **Condition 3** or **Condition 4**.

- **Pair query:** If a pair query involves a vertex in an already realized vertex group, then its answer is already determined and it does not reveal any unrealized groups. Otherwise, if a pair query only involves unrealized vertex groups, we show that the probability it reveals any unrealized groups is at most $\frac{8t}{n^{2\gamma}}$. First consider the case that the query is within a single

unrealized group S_i. For a $G' \sim \mathcal{D}_{\text{rz}}$, the probability that this query discovers a non-edge is at most $\frac{t}{\binom{n^\gamma}{2}}$. By Proposition 6.5.8, conditioned on S_i being matched to another S_j in the meta graph H, the probability that $G'[S_i \cup S_j]$ is consistent with previous answers is $\geqslant .99$. Therefore, this query discovers a non-edge with probability $\leqslant \frac{2t}{\binom{n^\gamma}{2}}$.

Then consider the case that the query is between two unrealized groups S_i, S_j. If S_i is not matched to S_j in the meta graph H, then the pair query does not discover an edge, since there is no edge between S_i, S_j. Otherwise, for a $G' \sim \mathcal{D}_{\text{rz}}$, conditioned on S_i being matched to S_j, the pair query discovers an edge with probability $\frac{2t}{n^{2\gamma}}$. By Proposition 6.5.8, the probability that $G'[S_i \cup S_j]$ is consistent with previous answers with probability $\geqslant .99$. Therefore, this query discovers an edge with probability $\leqslant \frac{4t}{n^{2\gamma}}$.

- **Neighbor query:** Consider a neighbor query on a vertex u in an unrealized vertex group S_i. For a $G' \sim \mathcal{D}_{\text{rz}}$, the query finds an edge going out of S_i with probability $\frac{t}{\binom{n^\gamma}{2}}$. By Proposition 6.5.8, conditioned on S_i being matched to another S_j in the meta graph H, the probability that $G'[S_i \cup S_j]$ is consistent with previous answers with probability $\geqslant .99$. Therefore, this query discovers an outgoing edge with probability $\leqslant \frac{2t}{\binom{n^\gamma}{2}}$.

Combining the above, a query makes an unrealized vertex group satisfy **Condition 3** or **Condition 4** with probability at most $\frac{8t}{n^{2\gamma}}$. Also, by doing so, a query can increase the number of realized vertex groups by at most 4. As a result, the expected increase in the number of realized groups that do not satisfy **Condition 1** or **Condition 2** over all queries made by $\mathcal{A}$ is at most

$$\begin{aligned}
4 \cdot \#\text{queries} \cdot \frac{8t}{n^{2\gamma}} =& \frac{32n^{\min\{1+\gamma, 2-2\gamma\}-\delta} \cdot t}{n^{2\gamma}} \\
=& \frac{32n^{\min\{1+\gamma, 2-2\gamma\}-\delta} n^{\max\{0, 3\gamma-1\}+\frac{1}{\sqrt{\log n}}}}{n^{2\gamma}} \qquad \text{(plugging in the value of } t\text{)} \\
=& 32n^{1-\gamma+\frac{1}{\sqrt{\log n}}-\delta} \\
\leqslant& n^{1-\gamma-\Omega(1)} \leqslant o(n^{1-\gamma}).
\end{aligned}$$

Then the proposition follows by an application of Chernoff bounds. □

Suppose we are now at the end of the algorithm $\mathcal{A}$. Let $\mathcal{D}_{\text{rz}}$ be $\mathcal{D}$ conditioned on all edge slots

incident on realized vertex groups, as defined above. Similarly, $G' \sim \mathcal{D}_{\text{rz}}$ is *not* necessarily consistent with the answers we gave to $\mathcal{A}$'s queries that involve unrealized vertex groups, albeit these answers are deterministic by Proposition 6.5.7. Also, let $\boldsymbol{a}$ denote the answers we gave to all queries made by $\mathcal{A}$, and let $\mathcal{D}_{\text{rz},\boldsymbol{a}}$ denote the conditional distribution of $\mathcal{D}_{\text{rz}}$ on $\boldsymbol{a}$.

Lemma 6.5.10. *Let $(S, \bar{S})$ be any fixed cut with $|S| \in [n/3, 2n/3]$. With probability at least $1 - 1/n$, the size of the cut $(S, \bar{S})$ in $G'' \sim \mathcal{D}_{\text{rz},\boldsymbol{a}}$ is at least $n^{2\gamma + \frac{1}{\sqrt{\log n}}}/10^7$.*

Proof. Suppose after $\mathcal{A}$ terminates, the number of realized vertex groups is bounded by $o(n^{1-\gamma})$, which by Proposition 6.5.9 happens with high probability. Suppose we generate a $G' \sim \mathcal{D}_{\text{rz}}$ using the process in Figure 6.2. Consider an S_i that is among the first unmatched $n^{1-\gamma}/13$ unrealized vertex groups that we iterate over at Step 4a in Figure 6.2. We claim that, with probability at least .1 over the choice of S_j matched to S_i and the edges we add between S_i, S_j, $t/100$ of the latter edges are across the cut $(S, \bar{S})$.

To prove the claim, note that the number of choices of S_j to be matched to S_i is at least $5n^{1-\gamma}/6$. Let U denote the vertices in these S_j's, and thus we have $|U| \geqslant 5n/6$. Define $T := S \cap U$ and $T' := \bar{S} \cap U$, which satisfy $|T| + |T|' = |U|$ and $|T| \in [|U|/6, 5|U|/6]$. Then the expected number of edge slots between S_i, S_j that are across the cut $(S, \bar{S})$ is given by

$$
\begin{aligned}
&\frac{1}{\#j\text{'s}} \sum_j |S_i \cap S||S_j \cap \bar{S}| + |S_i \cap \bar{S}||S_j \cap S| \\
=&\frac{1}{\#j\text{'s}} \left(|S_i \cap S| \cdot |T'| + |S_i \cap \bar{S}| \cdot |T|\right) \qquad \text{(moving the summation inside)} \\
\geqslant&\frac{1}{n^{1-\gamma}} \cdot \frac{|U|}{6} \left(|S_i \cap S| + |S_i \cap \bar{S}|\right) \qquad \text{(as } |T|, |T'| \geqslant |U|/6\text{)} \\
\geqslant&\frac{1}{n^{1-\gamma}} \cdot \frac{5n/6}{6} \cdot n^{\gamma} \qquad \text{(by } |U| \geqslant 5n/6\text{)} \\
>&.13n^{2\gamma}.
\end{aligned}
$$

Then the expected number of edges that we add between S_i, S_j that fall in these slots is at least

$$.13n^{2\gamma} \cdot \frac{2t}{n^{2\gamma}} = .26t.$$

Since the number of edges between S_i, S_j is $2t$, by Markov's inequality, the number such edges across the cut $(S, \bar{S})$ is at least $t/100$ with probability $\geqslant .1$, as desired.

Thus, for a $G' \in \mathcal{D}_{\text{rz}}$, in expectation, at least $n^{1-\gamma}/130$ of the S_i's satisfy that between S_i and the matched S_j, $t/100$ edges are across the cut $(S, \bar{S})$. By a Chernoff bound, with probability at least $1 - e^{-n^{1-\gamma}/500}$, the number of such S_i's is at least $n^{1-\gamma}/1300$, in which case the cut size of $(S, \bar{S})$ in G' is at least

$$\begin{aligned}\frac{t}{100} \cdot \frac{n^{1-\gamma}}{1300} =& 130000^{-1} n^{1-\gamma} n^{\max\{0,3\gamma-1\}+\frac{1}{\sqrt{\log n}}} \\ \geqslant& 10^{-7} n^{2\gamma+\frac{1}{\sqrt{\log n}}}.\end{aligned}$$

On the other hand, by Proposition 6.5.8, G' is consistent with all answers $\boldsymbol{a}$ that we gave to $\mathcal{A}$ with probability at least

$$.998^{n^{1-\gamma}/2} \geqslant e^{-.0015 n^{1-\gamma}}.$$

As a result, the cut $(S, \bar{S})$ in $G'' \sim \mathcal{D}_{\text{rz},\boldsymbol{a}}$ has size at least $10^{-7} n^{2\gamma+\frac{1}{\sqrt{\log n}}}$ with probability at least $1 - e^{-0.0005 n^{1-\gamma}}$, which suffices for proving the lemma. □

We now conclude this section by proving Theorem 6.5.3.

Proof of Theorem 6.5.3. Let $\mathcal{A}$ be a deterministic algorithm that makes $n^{\min\{1+\gamma, 2-2\gamma\}-\delta}$ queries for some constant $\delta > 0$. Suppose for the sake of contradiction, on an input graph $G \sim \mathcal{D}$, $\mathcal{A}$ outputs with probability $\Omega(1)$ a polylog(n)-approximate hierarchical clustering tree. First, we turn this tree into a full binary tree such that the bi-partition of each internal node is $[1/3, 2/3]$-balanced, while increasing the cost by at most an $O(1)$ factor. We then consider the bi-partition of the root, which is a cut $(S, \bar{S})$ with $|S| \in [n/3, 2n/3]$. By Lemma 6.5.10, conditioned on the answers $\mathcal{A}$ got, this cut has size at least $n^{2\gamma+\frac{1}{\sqrt{\log n}}}/10^7$ with high probability. However, by Proposition 6.5.5, the cost of the optimal hierarchical clustering tree of G is at most $O(n^{1+2\gamma})$. This means that $\mathcal{A}$ only obtains an $n^{o(1)}$-approximation with high probability, a contradiction. □

6.6. Experimental Results

We now present our experimental results. The goal of our experiments is to test our data/graph sparsification framework (Algorithm 6.4.3) which forms the basis of all our algorithms under the different models of computation. We ask the following question: if we run an HC algorithm on the sparsified graph produced by Algorithm 6.4.3, then what will be the loss in the quality of solution compared to the same HC algorithm run on the original graph? We would also like to understand the tradeoff between the sparsification rate and the solution quality of certain HC algorithms.

We test our data sparsification algorithm on two large real-world datasets Boston and Newsgroup, which are both part of the standard Scikit-learn library and were used in previous HC papers [CKM17, RP17]. We calculate the similarity between two data points using the Gaussian kernel as in [RP17, MS21] and only insert an edge if its weight (i.e. similarity between its two endpoints) is bigger than 10^{-10}.

Since we are not aware of any existing implementation of the recursive sparsest cut algorithm, we choose to test two linkage algorithms, namely average linkage and complete linkage, which, as shown in [MS21], are among the best-performed algorithms on the two datasets we consider. We run these two algorithms both on the entire graph and on the sparsified graphs obtained by our Algorithm 6.4.3, and compare the cost of the HC we get in each case, where the cost is measured using Dasgupta's objective function (Eq. (1.2)). When running our Algorithm 6.4.3, we also try different sampling rates to obtain subgraphs of various edge densities. Note that, we always measure the cost of an HC in the original graph as opposed to in the subsampled graph.

We present our experimental results in the tables below. Here we write absolute HC cost to denote the cost of the HC obtained by running the corresponding algorithm on the entire graph, and write relative HC cost to denote the cost of the HC obtained by running the algorithm on the subsampled graph divided by the absolute HC cost.

For the Boston dataset, one can see that even when we only sample $<5\%$ of the edges, we already get a very good relative HC cost of 1.15; for the Newsgroup dataset, sampling only $<0.5\%$ of the edges already gives us a low relative HC cost of 1.188 for average linkage and 1.05 for complete

Table 6.1: Performance of average linkage on Boston dataset (506 vertices, 95566 edges), with absolute HC cost 186085.271.

Number (Percentage) of Sampled Edges	Relative HC Cost
4040 (4.23%)	1.146
5861 (6.13%)	1.068
7010 (7.34%)	1.043
9018 (9.44%)	1.016
9756 (10.21%)	1.012
11941 (12.50%)	1.000

Table 6.2: Performance of complete linkage on Boston dataset (506 vertices, 95566 edges), with absolute HC cost 226485.146.

Number (Percentage) of Sampled Edges	Relative HC Cost
4032 (4.22%)	1.158
5885 (6.16%)	1.135
7002 (7.33%)	1.113
9033 (9.45%)	1.066
9773 (10.23%)	1.037
11900 (12.45%)	1.006

linkage. As the number of sampled edges grow, the relative HC cost becomes smaller and gets closer to 1.

Table 6.3: Performance of average linkage on Newsgroup dataset (3516 vertices, 6178853 edges), with absolute HC cost 3960145237.881.

Number (Percentage) of Sampled Edges	Relative HC Cost
28495 (0.46%)	1.188
55116 (0.89%)	1.169
103804 (1.68%)	1.143
203095 (3.29%)	1.118
358440 (5.80%)	1.111
1029648 (16.66%)	1.068

Table 6.4: Performance of complete linkage on Newsgroup dataset (3516 vertices, 6178853 edges), with absolute HC cost 3960674776.482.

Number (Percentage) of Sampled Edges	Relative HC Cost
28476 (0.46%)	1.050
55042 (0.89%)	1.010
103756 (1.68%)	1.001
203323 (3.29%)	1.000
358446 (5.80%)	1.000
1029274 (16.66%)	1.000

APPENDIX A

MISSING PROOFS IN CHAPTER 3

A.1. Proof of Proposition 2.2.4

Proposition A.1.1. *Given query access to the adjacency matrix, the algorithm in Proposition 2.2.3 can be implemented in* $O(n \cdot Q(t, 1/\gamma) \log n)$ *space and* $\mathrm{poly}(n, Q(t, 1/\gamma))$ *time.*

Proof Sketch. We now briefly describe how the algorithm in Proposition 2.2.3 can be implemented in a space-efficient manner, given query access to the adjacency matrix of the underlying graph G. Roughly speaking, the algorithm works as follows. Initially, it starts with an arbitrary equitable t-partition. As long as more than γ fraction of the C_i, C_j pairs are not γ-regular, for each such irregular pair C_i, C_j, a *witnessing pair* $X \subseteq C_i, Y \subseteq C_j$ is identified such that X, Y violate the regularity property. Then the algorithm does a refinement of the partition such that simultaneously for all irregular pairs C_i, C_j with witnessing pair X, Y, vertices in X vs. $C_i \setminus X$ and those in Y vs. $C_j \setminus Y$ are separated. A potential function argument then shows that there cannot be more than $\mathrm{poly}(t, 1/\gamma)$ refinements before we obtain a γ-regular partition.

Note here that the refinement of the partition is easy to implement in $O(nk \log n)$ space and $\mathrm{poly}(n)$ time, with k being the number of classes, as it only requires storing a description of the vertex partition. It then remains to analyze the process of finding a witness X, Y for each irregular pair C_i, C_j. This is done in [Alo02] by an approximation algorithm, where the main step requires computing the number of common neighbors for each vertex pair $u, v \in V$, by squaring the adjacency matrix via fast matrix multiplication. However, this can be easily done in $O(n \log n)$ space and $\mathrm{poly}(n)$ time given query access to the adjacency matrix. Thus the entire algorithm can be implemented in $O(n \cdot Q(t, 1/\gamma) \log n)$ space and $\mathrm{poly}(n, Q(t, 1/\gamma))$ time. □

APPENDIX B

MISSING PROOFS IN CHAPTER 4

B.1. A hard instance for decomposing matrix-weighted graphs into large subgraphs without small nonzero eigenvalues

The proof of the following proposition appears in Appendix B.3 (in "Proof of Proposition 4.5.11").

Proposition B.1.1. *For a vector $x \in \mathbb{R}^{nk}$, we have*

$$x^T L x = \sum_{u \sim v} (x_u - x_v)^T \phi_{uv} \phi_{uv}^T (x_u - x_v).$$

By the above proposition, we have:

Proposition B.1.2. *For any nk-dimensional vector x such that $x_u = x_v \in \mathbb{R}^k$ for any $u, v \in V$, we have $x^T L x = 0$. As a result, the null space of L has dimension at least k.*

First consider the following n-vertex 2×2 matrix-weighted graph, call it R, whose vertices can be embedded into points that are uniformly distributed along a unit 2D-circle. Specifically, for each vertex $u \in \{1..n\}$, assign to it a coordinate

$$c_u \stackrel{\text{def}}{=} (\cos \frac{2u\pi}{n}, \sin \frac{2u\pi}{n})^T.$$

Then for every $u \neq v$, connect them by an edge weighted by (the outer product of) the segment connecting c_u, c_v. That is, add an edge (u, v) with weight $(c_u - c_v)(c_u - c_v)^T$. Then one can see that the Laplacian matrix (and thus also the normalized Laplacian matrix) of graph R has three zero eigenvalues — apart from the two trivial zero eigenvectors mentioned in Proposition B.1.2, one can also construct another zero eigenvector x where x_u equals c_u rotated by 90 degrees counter-clockwise around the origin, namely $x_u = (-\sin \frac{2u\pi}{n}, \cos \frac{2u\pi}{n})^T$. Then we have for any $u \neq v$ that $x_u - x_v$ is orthogonal to ϕ_{uv}, and hence

$$x^T L_R x = \sum_{\{u,v\}} (x_u - x_v)^T \phi_{uv} \phi_{uv}^T (x_u - x_v) = 0.$$

Now consider creating a noisy version of the graph R, call it $\mathcal{R}$, as follows. Between every pair of vertices u, v, we add a small stochastic noise on the weight of the edge connecting them. Specifically, let $\tilde{\phi}_{uv}$'s be drawn independently from the 2-dimensional Gaussian $\mathcal{N}(0, \varepsilon I)$, for some sufficiently small ε (say $o(1/\text{poly}(n))$). Then we let the edge weight of (u, v) be

$$(\phi_{uv} + \tilde{\phi}_{uv})(\phi_{uv} + \tilde{\phi}_{uv})^T.$$

Now for the vector x with $x_u = (-\sin\frac{2u\pi}{n}, \cos\frac{2u\pi}{n})^T$, we have

$$\begin{aligned}\mathbb{E}_{\tilde{\phi}_{uv}}\left[x^T L_{\mathcal{R}} x\right] &= \sum_{\{u,v\}} \mathbb{E}_{\tilde{\phi}_{uv}}\left[(x_u - x_v)^T(\phi_{uv} + \tilde{\phi}_{uv})(\phi_{uv} + \tilde{\phi}_{uv})^T(x_u - x_v)\right] \\ &= \sum_{\{u,v\}} \mathbb{E}_{\tilde{\phi}_{uv}}\left[(x_u - x_v)^T \tilde{\phi}_{uv}\tilde{\phi}_{uv}^T(x_u - x_v)\right] = \Theta(\varepsilon n^2).\end{aligned}$$

Also note that $\|x\|^2 = n$, and x is orthogonal to the two trivial zero eigenvectors of $L_{\mathcal{R}}$ as $\sum_u x_u = 0$. Therefore $L_{\mathcal{R}}$'s smallest nonzero eigenvalue is at most $o(1/\text{poly}(n))$. One can verify that in $\mathcal{R}$ the degree of each vertex u has a constant ratio between its largest and smallest eigenvalues, and as a result the normalized Laplacian $N_{\mathcal{R}}$ also has a nonzero eigenvalue bounded by $o(1/\text{poly}(n))$. One can similarly show the same property for every large enough subgraph of $\mathcal{R}$ (say, e.g., with $\Omega(n^2)$ edges).

B.2. Missing proofs from Section 4.2

Proof of proposition 4.2.2. By Proposition 4.2.1, with probability $1 - 1/n$, the effective resistance between $\pi(1)$ and $\pi(n/2+1)$ is at least $1/48$. Thus whenever the crossing edge has positive weight at least $\frac{1}{4}$, one can tell if the crossing edge is present or not by looking at a 1.0001-spectral sparsifier of the graph. On the other hand, if we do add the crossing edge, it has weight $\geqslant \frac{1}{4}$ with probability at least 0.4, by properties of a standard Gaussian. Therefore if a linear sketch can compute a 1.0001-spectral sparsifier with probability 0.9, then it can decide if the crossing edge is present with

probability

$$\geqslant (1 - 1/n)(0.5 + 0.5 \cdot 0.4)0.9 \geqslant 0.6.$$

□

Proof of Proposition 4.2.3. For the i^{th} row of the sketching matrix Φ, if its support is on edges incident on vertex u, we say the i^{th} row belongs to u. Let $U \subseteq V$ be the subset of vertices u such that the number of rows of Φ that belong to u is at most $\frac{1000N}{n}$. By Markov's inequality we have $|U| \geqslant 0.999n$. Let $F = \binom{U}{2}$ be the edge slots within U. Then $|F| \geqslant 0.998\binom{n}{2}$.

Let $(\Phi w)_{\pi,\text{yes}}$ be Φw conditioned on π and the presence of the crossing edge; similarly let $(\Phi w)_{\pi,\text{no}}$ be Φw conditioned on π and the absence of the crossing edge. Let $\phi_{uv} \in \mathbb{R}^N$ be the column of Φ corresponding to edge slot (u, v). By Theorem 2.3.10, for any π we have

$$d_{\text{TV}}\left((\Phi w)_{\pi,\text{yes}}, (\Phi w)_{\pi,\text{no}}\right) \leqslant$$
$$O(1) \cdot \min\left\{1, \phi^T_{\pi(1)\pi(n/2+1)} \left(\sum_{\text{non-crossing } (u,v)} n^{4/5} \log^{-1} n \phi_{uv}\phi^T_{uv}\right)^\dagger \phi_{\pi(1)\pi(n/2+1)}\right\}. \quad \text{(B.1)}$$

Note that we have slightly abused the notation † in the second term on the RHS of (B.1): in the case that $\phi_{\pi(1)\pi(n/2+1)}$ is not in the range of the matrix inside the summation, the term should be understood as infinity.

By restricting the summation on the RHS to non-crossing edges in F, we have, by Fact 2.3.6, for any π

$$\phi^T_{\pi(1)\pi(n/2+1)} \left(\sum_{\text{non-crossing } (u,v)} n^{4/5} \log^{-1} n \phi_{uv}\phi^T_{uv}\right)^\dagger \phi_{\pi(1)\pi(n/2+1)} \leqslant$$
$$\phi^T_{\pi(1)\pi(n/2+1)} \left(\sum_{\text{non-crossing } (u,v)\in F} n^{4/5} \log^{-1} n \phi_{uv}\phi^T_{uv}\right)^\dagger \phi_{\pi(1)\pi(n/2+1)}. \quad \text{(B.2)}$$

Now consider a sketching matrix $\Phi'' \in \mathbb{R}^{N \times \binom{n}{2}}$ that is obtained from Φ by zeroing out the columns

corresponding to edge slots *not* in F. Then we have by (B.2) that for any π such that $(\pi(1), \pi(n/2+1)) \in F$ (thus the column of Φ'' corresponding to the crossing edge is *not* zeroed out)

$$d_{\mathrm{TV}}\left((\Phi w)_{\pi,\mathrm{yes}}, (\Phi w)_{\pi,\mathrm{no}}\right) \leqslant d_{\mathrm{TV}}\left((\Phi'' w)_{\pi,\mathrm{yes}}, (\Phi'' w)_{\pi,\mathrm{no}}\right).$$

Using the fact that $\Pr_\pi\left[(\pi(1), \pi(n/2+1)) \in F\right] = |F|/\binom{n}{2} \geqslant 0.998$, we have

$$\mathbb{E}_\pi\left[d_{\mathrm{TV}}\left((\Phi'' w)_{\pi,\mathrm{yes}}, (\Phi'' w)_{\pi,\mathrm{no}}\right)\right] \geqslant \mathbb{E}_\pi\left[d_{\mathrm{TV}}\left((\Phi w)_{\pi,\mathrm{yes}}, (\Phi w)_{\pi,\mathrm{no}}\right)\right] - 0.002,$$

and as a result, there exists an incidence sketch using Φ'' that can distinguish between the Yes and No distributions with probability 0.55. We then show that we can simulate such an incidence sketch that uses Φ'' by a signed sketch with a sketching matrix $\Phi' \in \mathbb{R}^{k \times \binom{n}{2}}$ such that $k \leqslant O(1) \cdot \max\left\{1, \frac{N \log n}{n}\right\}$. Let $R_u \subseteq \{1, \ldots, N\}$ denote the indices of the rows that belong to u. Then we know $|R_u| \leqslant 1000N/n$ for all $u \in U$. It suffices to construct a Φ' from which we can recover, for each vertex $u \in U$, the sketch $\sum_{v \in U \setminus \{u\}} w_{uv} \cdot (\phi_{uv})_{R_u} \in \mathbb{R}^{|R_u|}$. Now consider sketching matrices $\Phi'_1, \ldots, \Phi'_{10 \log n} \in \mathbb{R}^{\frac{1000N}{n} \times \binom{n}{2}}$, which are initially set to be all-zero. We then assign values to these matrices as follows. For each $i = 1, \ldots, 10 \log n$, we generate a subset of vertices S_i by including each vertex independently in S_i with probability $1/2$. By Chernoff bound, we know that with high probability, we have that for every vertex u and every incident edge (u, v), $u \in S_i, v \in V - S_i$ holds for at least one i. Let i_{uv} be the smallest i such that $u \in S_i, v \in V - S_i$. Then for each vertex u and each incident edge $(u, v) \in F$, we let the column of $\Phi'_{i_{uv}}$ corresponding to (u, v) be

$$\Phi'_{i_{uv}}(:, (u, v)) = \begin{cases} (\phi_{uv})_{R_u} & u \text{ is } (u, v)\text{'s head} \\ -(\phi_{uv})_{R_u} & u \text{ is } (u, v)\text{'s tail,} \end{cases}$$

where we augment $(\phi_{uv})_{R_u}$ by 0's if $|R_u| < \frac{1000N}{n}$. Then for each vertex $u \in S_i$, we have

$$\left(\Phi'_i B^w\right)(:, u) = \sum_{v \in U : i_{uv} = i} w_{uv} \cdot (\phi_{uv})_{R_u},$$

and therefore

$$\sum_{i=1}^{10\log n} \left(\Phi_i' B^w\right)(:,u) = \sum_{v\in U\setminus\{u\}} w_{uv}\cdot(\phi_{uv})_{R_u}.$$

This implies that we can construct a desired Φ' by stacking up the Φ_i''s:

$$\Phi' = \begin{pmatrix} \Phi_1' \\ \vdots \\ \Phi_{10\log n}' \end{pmatrix} \in \mathbb{R}^{\frac{10000N\log n}{n}\times\binom{n}{2}}.$$

□

Proof of Proposition 4.2.4. Consider fixing a sketching matrix $\Phi \in \mathbb{R}^{k\times\binom{n}{2}}$ and a permutation $\pi : \{1..n\} \to \{1..n\}$. We are interested in the total variation distance between $(\Phi B^w)_{\pi,\text{yes}}$ and $(\Phi B^w)_{\pi,\text{no}}$. Let us also fix a vertex $u \in \{1..n\}$ and consider the u-th column $(\Phi B^w)(:,u) \in \mathbb{R}^k$ of the sketch obtained. We have (recall that ϕ_e is the e-th column of Φ)

$$(\Phi B^w)(:,u) = \sum_e B^w_{eu}\phi_e. \tag{B.3}$$

If e is not present in the graph, or if e is not incident on u, then the entry B^w_{eu} is zero. Otherwise (e present and incident on u), we know that B^w_{eu} follows a univariate Gaussian distribution $\mathcal{N}(\mu_{e,u},\sigma^2_{e,u})$, where for non-crossing e

$$\mu_{e,u} = \begin{cases} 8n^{2/5} & u \text{ is } e\text{'s head} \\ -8n^{2/5} & u \text{ is } e\text{'s tail} \end{cases} \qquad \text{and} \qquad \sigma^2_{e,u} = n^{4/5}\log^{-1} n \tag{B.4}$$

and for the crossing edge e

$$\mu_{e,u} = 0 \qquad \text{and} \qquad \sigma^2_{e,u} = 1. \tag{B.5}$$

Therefore, $B^w_{eu}\phi_e$ follows a k-dimensional Gaussian distribution:

$$B^w_{eu}\phi_e \sim \mathcal{N}(\mu_{e,u}\phi_e, \sigma^2_{e,u}\phi_e\phi_e^T). \tag{B.6}$$

Also notice that the distributions of B^w_{eu}'s are independent for different edges. Thus by Fact 2.3.9

$$(\Phi B^w)(:,u) \sim \mathcal{N}\left(\sum_{e\sim u}\mu_{e,u}\phi_e, \sum_{e\sim u}\sigma^2_{e,u}\phi_e\phi_e^T\right) \tag{B.7}$$

As a result, the sketch ΦB^w can be seen as an nk-dimensional Gaussian.

We next consider the correlations between different columns of ΦB^w. For two vertices u, v, if there is no edge between them, then the distributions of $(\Phi B^w)(:,u)$ and $(\Phi B^w)(:,v)$ are independent, so the correlation between them is zero. Otherwise (if the edge $e = (u,v)$ is present), the only correlation between $(\Phi B^w)(:,u)$ and $(\Phi B^w)(:,v)$ is the one between $B^w_{eu}\phi_e$ and $B^w_{ev}\phi_e$, who are negations of each other. Therefore the correlation between $(\Phi B^w)(:,u)$ and $(\Phi B^w)(:,v)$ is just the negative covariance of $B^w_{eu}\phi_e$:

$$\mathbb{E}\left[\ (\,(\Phi B^w)(:,u) - \mathbb{E}\left[(\Phi B^w)(:,u)\right]\,)\,(\,(\Phi B^w)(:,v) - \mathbb{E}\left[(\Phi B^w)(:,v)\right]\,)^T\ \right] = -\sigma^2_{e,u}\phi_e\phi_e^T. \tag{B.8}$$

We are now ready to write down the covariance matrix of the sketch ΦB^w. First consider the covariance matrix of $(\Phi B^w)_{\pi,\text{no}}$, call it $\Sigma_{(\Phi B^w)_{\pi,\text{no}}} \in \mathbb{R}^{nk\times nk}$. We can write $\Sigma_{(\Phi B^w)_{\pi,\text{no}}}$ as an $n\times n$ block matrix (with block size $k\times k$) by

$$\left(\Sigma_{(\Phi B^w)_{\pi,\text{no}}}\right)_{uv} = \begin{cases} \sum_{\text{non-crossing } e\sim u} n^{4/5}\log^{-1} n\phi_e\phi_e^T & u = v \\ -n^{4/5}\log^{-1} n\phi_e\phi_e^T & u\neq v \text{ and } e=(u,v) \text{ is non-crossing} \\ 0 & \text{otherwise.} \end{cases}$$

One can also verify that $\Sigma_{(\Phi B^w)_{\pi,\text{no}}} = \sum_{\text{non-crossing}(u,v)} n^{4/5}\log^{-1} n b_{uv}b_{uv}^T$, where b_{uv}'s are defined in (4.1). That is $\Sigma_{(\Phi B^w)_{\pi,\text{no}}}$ is exactly the matrix L_π.

If the crossing edge is present, using the fact that its weight follows a standard Gaussian distribution,

the covariance matrix can be written as

$$\Sigma_{(\Phi B^w)_{\pi,\text{yes}}} = \Sigma_{(\Phi B^w)_{\pi,\text{no}}} + b_{\pi(1)\pi(n/2+1)} b^T_{\pi(1)\pi(n/2+1)} \tag{B.9}$$

Notice that, since the crossing edge's weight has zero mean, $(\Phi B^w)_{\pi,\text{yes}}$ and $(\Phi B^w)_{\pi,\text{no}}$ have the same mean. Therefore, we can invoke Theorem 2.3.10 and obtain an upper bound (up to a constant factor) on the total variation distance between the two distributions as

$$O(1) \cdot \min\left\{1, b^T_{\pi(1)\pi(n/2+1)} \Sigma^{\dagger}_{(\Phi B^w)_{\pi,\text{no}}} b_{\pi(1)\pi(n/2+1)}\right\}, \tag{B.10}$$

as desired. □

Proof of Claim 4.2.8. Let $n' = |U| \geqslant d_{\min} \geqslant \frac{|E_\phi|}{8n}$ be the number of vertices in I. Notice that by conditioning on an $f \in I$ being the crossing edge, we have $\pi(1), \pi(n/2+1) \in U$ both. Let U_i denote the vertices in U that are in the i^{th} block of the cycle. We have with high probability over π that each $|U_i| \in [\frac{n'}{2n^{4/5}}, \frac{2n'}{n^{4/5}}]$. By Lemma 4.2.7, with probability at least $1 - 1/n^4$ over π, each $I[U_i \cup U_{i+1}]$ is a $\frac{1}{n^{o(1)}}$-expander with minimum degree $\geqslant \frac{|E_\phi|}{16n^{9/5}}$. Now consider constructing a graph $\tilde{I}$ supported on U by, for each $0 \leqslant i < \ell$, adding a clique of weight $\frac{|E_\phi|}{n'n}$ on $U_i \cup U_{i+1}$ (whose vertex degrees are all $\Theta(\frac{|E_\phi|}{n^{9/5}})$). One can show that in the graph $\tilde{I}$, the effective resistance between $\pi(1), \pi(n/2+1)$ is maximized when we set n' to be its minimum $\Theta(\frac{|E_\phi|}{n})$, in which case the effective resistance is $\Theta(n^{4/5}) \cdot \left(\frac{n^{9/5}}{|E_\phi|}\right)^2$. Then by the expander property, the effective resistance between $\pi(1), \pi(n/2+1)$ in H_π is at most $\Theta(n^{4/5+o(1)}) \cdot \left(\frac{n^{9/5}}{|E_\phi|}\right)^2$. Finally, since $\mathcal{H}_\pi$ is $n^{4/5} \log^{-1} n H_\pi$, we have that the effective resistance between $\pi(1), \pi(n/2+1)$ in $\mathcal{H}_\pi$ is at most $\Theta(n^{o(1)}) \cdot \left(\frac{n^{9/5}}{|E_\phi|}\right)^2$, which is bounded by u_f. □

Proof of Proposition 4.2.18. Suppose w.l.o.g. $\lambda_1, \ldots, \lambda_\ell$ are all eigenvalues between $(0, \zeta]$. Then

$$\begin{aligned}
&\sum_{(u,v)\in E} \left(D^{\dagger/2} b_{uv}\right)^T \left(\sum_{i=1}^{\ell} \lambda_i f_i f_i^T\right)^\dagger D^{\dagger/2} b_{uv} \\
=&\sum_{(u,v)\in E} \operatorname{Tr}\left(\left(\sum_{i=1}^{\ell} \lambda_i f_i f_i^T\right)^\dagger D^{\dagger/2} b_{uv} b_{uv}^T D^{\dagger/2}\right) \\
=&\operatorname{Tr}\left(\left(\sum_{i=1}^{\ell} \lambda_i f_i f_i^t\right)^\dagger \left(\sum_{(u,v)\in E} D^{\dagger/2} b_{uv} b_{uv}^T D^{\dagger/2}\right)\right) \\
=&\operatorname{Tr}\left(\left(\sum_{i=1}^{\ell} \lambda_i f_i f_i^t\right)^\dagger N_G\right) \\
=&\operatorname{Tr}\left(\left(\sum_{i=1}^{\ell} \lambda_i f_i f_i^t\right)^\dagger \left(\sum_{i=1}^{n} \lambda_i f_i f_i^T\right)\right) \\
=&\operatorname{Tr}\left(\left(\sum_{i=1}^{\ell} \lambda_i f_i f_i^t\right)^\dagger \left(\sum_{i=1}^{\ell} \lambda_i f_i f_i^T\right)\right) = \ell,
\end{aligned}$$

as desired. □

B.3. Missing proofs from Section 4.5

Proof of Proposition 4.5.11. The claim follows by

$$\begin{aligned}
x^T L x =& x^T \left(\sum_{u\sim v} b_{uv} b_{uv}^T\right) x \\
=& \sum_{u\sim v} \langle x, b_{uv}\rangle^2 \\
=& \sum_{u\sim v} \left(\langle x_u, \phi_{uv}\rangle - \langle x_v, \phi_{uv}\rangle\right)^2 \\
=& \sum_{u\sim v} \langle x_u - x_v, \phi_{uv}\rangle^2 .
\end{aligned}$$

□

Proof of Proposition 4.5.15. The claim follows by

$$\begin{aligned} x^T N x =& x^T \left(\sum_{u \sim v} (D^{\dagger/2} b_{uv})(D^{\dagger/2} b_{uv})^T \right) x \\ =& \sum_{u \sim v} \left\langle x, D^{\dagger/2} b_{uv} \right\rangle^2 \\ =& \sum_{u \sim v} \left(\left\langle x_u, D_u^{\dagger/2} \phi_{uv} \right\rangle - \left\langle x_v, D_v^{\dagger/2} \phi_{uv} \right\rangle \right)^2 . \end{aligned}$$

□

Proof of Proposition 4.5.16. The lower bound of 0 follows from that N is positive semi-definite. We then prove the upper bound by showing that for any $x \in \mathbb{R}^{nk}$, its Rayleigh quotient $R(x) = \frac{x^T N x}{x^T x}$ is at most 2. It is then equivalent to show that $x^T N x \leqslant 2 \|x\|^2$:

$$\begin{aligned} x^T N x =& \sum_{u \sim v} \left(\left\langle x_u, D_u^{\dagger/2} \phi_{uv} \right\rangle - \left\langle x_v, D_v^{\dagger/2} \phi_{uv} \right\rangle \right)^2 \\ \leqslant& \sum_{u \sim v} \left(2 \left\langle x_u, D_u^{\dagger/2} \phi_{uv} \right\rangle^2 + 2 \left\langle x_v, D_v^{\dagger/2} \phi_{uv} \right\rangle^2 \right) \\ =& 2 \sum_u \sum_{v \sim u} \left\langle x_u, D_u^{\dagger/2} \phi_{uv} \right\rangle^2 \\ =& 2 \sum_u \sum_{v \sim u} x_u^T D_u^{\dagger/2} \phi_{uv} \phi_{uv}^T D_u^{\dagger/2} x_u \\ =& 2 \sum_u x_u^T D_u^{\dagger/2} \left(\sum_{v \sim u} \phi_{uv} \phi_{uv}^T \right) D_u^{\dagger/2} x_u \\ =& 2 \sum_u x_u^T D_u^{\dagger/2} D_u D_u^{\dagger/2} x_u \\ \leqslant& 2 \sum_u x_u^T x_u = 2 \sum_u \|x_u\|^2 = 2 \|x\|^2 . \end{aligned}$$

Here the last inequality follows from that $D_u^{\dagger/2} D_u D_u^{\dagger/2} = \Pi_{D_u} \preceq I$. □

B.4. Missing proofs from Section 4.8

Proof of Lemma 4.8.4. Define $Y = D^{1/2}X$. Then

$$\min_{X \in \mathcal{X}} \frac{\det\left(X^T L X\right)}{\det\left(X^T D X\right)} = \min_{D^{\dagger/2}Y \in \mathcal{X}} \frac{\det\left(Y^T D^{\dagger/2} L D^{\dagger/2} Y\right)}{\det\left(Y^T Y\right)}. \tag{B.11}$$

Let the eigenvalues of $Y^T Y$ be $\nu_1, \ldots, \nu_\ell$ and let $g_1, \ldots, g_\ell \in \mathbb{R}^\ell$ be a corresponding set of orthonormal eigenvectors. We also know that YY^T has eigenvalues $\nu_1, \ldots, \nu_\ell, 0, \ldots, 0$. Let $h_1, \ldots, h_{nk} \in \mathbb{R}^{nk}$ be a corresponding set of orthonormal eigenvectors of YY^T. Then

$$\det(Y^T Y) = \Pi_{i=1}^{\ell} \nu_i$$

and

$$\begin{aligned}
\det\left(Y^T D^{\dagger/2} L D^{\dagger/2} Y\right) &= \det{}_+\left(L^{1/2} D^{\dagger/2} Y Y^T D^{\dagger/2} L^{1/2}\right) \\
&= \det{}_+\left(L^{1/2} D^{\dagger/2} \left(\sum_{i=1}^{\ell} \nu_i h_i h_i^T\right) D^{\dagger/2} L^{1/2}\right) \\
&= \det{}_+\left(L^{1/2} D^{\dagger/2} H V H^T D^{\dagger/2} L^{1/2}\right) \\
&= \det\left(V^{1/2} H^T D^{\dagger/2} L D^{\dagger/2} H V^{1/2}\right) \\
&= \left(\Pi_{i=1}^{\ell} \nu_i\right) \det\left(H^T D^{\dagger/2} L D^{\dagger/2} H\right),
\end{aligned}$$

where we define $H = \begin{pmatrix} h_1 & \ldots & h_\ell \end{pmatrix} \in \mathbb{R}^{nk \times \ell}$ and $V = \operatorname{diag}(\nu_1, \ldots, \nu_\ell) \in \mathbb{R}^{\ell \times \ell}$. The claim in the lemma then follows by noting that $\det\left(H^T D^{\dagger/2} L D^{\dagger/2} H\right)$ is minimized (over all H's with orthonormal columns in the range of N) when H's columns are bottom nonzero eigenvectors of $N = D^{\dagger/2} L D^{\dagger/2}$, a result of Cauchy interlacing. Therefore (B.11) equals $\det_\ell(N)$. □

Proof of Lemma 4.8.8. Consider the following process for transitioning G to G^s, where we use t to denote the current scaling, which is all one in the beginning.

1. Initially, let $t_e \leftarrow 1$ for all $e \in E$.
2. While $t \neq s$:

(a) Let $F \leftarrow \{e : t_e > s_e\}$, and let $\eta \leftarrow \min_{e \in F} t_e/s_e$.

(b) For each $e \in F$, let $t_e \leftarrow t_e/\eta$.

Note that this process terminates in finite time as in each loop we make $t_e = s_e$ for at least one extra e.

Claim B.4.1. *At the beginning of every while loop, we have for any e such that $s_e < t_e$*

$$R^t(e) \leqslant \frac{2\gamma k}{n}.$$

Proof. We prove this by induction. In the beginning of the process, by γ-regularity, we have $R^t(e) \leqslant \frac{2\gamma k}{n}$ for all $e \in E$. Then for the induction step, note that in each while loop, we downscale the weights of *all* edges with $s_e < t_e$ by a same amount. From the point view of leverage scores, this is equivalent to increasing the weights of all other edges by a same multiple. As a result, the leverage scores of *all* edges with $s_e < t_e$ can only decrease (by Fact 2.3.6), and thus we have $R^t(e) \leqslant \frac{2\gamma k}{n}$ for these edges throughout. □

Claim B.4.2. *Consider fixing an iteration of the while loop and letting t and t' be the scaling before and after the execution of Line 2b respectively. Then we have*

$$\det_+(D^{t'}) \geqslant \left(1 - \frac{2\gamma k}{n}\right)^{2|F| \log \eta} \det_+(D^t).$$

Proof. For an $x \in [1, \ln \eta]$, define $t^x : E \to [0, 1]$ by

$$t_e^x = \begin{cases} e^{-x} \cdot t_e & e \in F \\ t_e & \text{o.w.} \end{cases}$$

Then we have $t' = t^{\ln \eta}$. Consider differentiating $\ln \det_+(D^{t^x})$:

$$\begin{aligned}
\frac{\mathrm{d} \ln \det_+(D^{t^x})}{\mathrm{d}x} =& \mathrm{Tr}\left(\left(D^{t^x}\right)^\dagger \frac{\mathrm{d}D^{t^x}}{\mathrm{d}x}\right) \\
=& -2\mathrm{Tr}\left(\left(D^{t^x}\right)^\dagger \left(D^{t^x}\right)^F\right) \\
=& -2\mathrm{Tr}\left(\left(D^{t^x}\right)^\dagger \left(\sum_{e\in F}\left(e^{t^x}_{u\leftarrow v}\left(e^{t^x}_{u\leftarrow v}\right)^T + e^{t^x}_{v\leftarrow u}\left(e^{t^x}_{v\leftarrow u}\right)^T\right)\right)\right) \\
=& -\sum_{e\in F} 2R^{t^x}(e) \geqslant -2|F| \cdot \frac{2\gamma k}{n}.
\end{aligned}$$

Therefore

$$\ln \det_+(D^{t'}) - \ln \det_+(D^t) = \int_0^{\ln \eta} \frac{\mathrm{d} \ln \det_+(D^{t^x})}{\mathrm{d}x} \mathrm{d}x \geqslant -|F| \cdot \frac{4\gamma k}{n} \cdot \ln \eta.$$

Thus we have

$$\frac{\det_+(D^{t'})}{\det_+(D^t)} \geqslant e^{-\frac{2\gamma k}{n}\cdot 2|F|\cdot \ln \eta} \geqslant \left(1 - \frac{2\gamma k}{n}\right)^{2|F|\cdot \ln \eta} \geqslant \left(1 - \frac{2\gamma k}{n}\right)^{2|F|\cdot \log \eta}$$

as desired. □

The lemma is then a direct consequence of the above claim. □

Proof of Lemma 4.8.9. Let s' be the scaling obtained at the end of the t^{th} iteration of the outermost while loop. Let t_1 be the total number of iterations executed so far by the first inner while loop (at Lines 15-16). Then by matrix determinant lemma

$$\begin{aligned}
\frac{\det_+(D^{s'})}{\det_+(D^{s^0})} &\leqslant \left(1 - \frac{3\gamma_2 \cdot k}{8n}\right)^{2t_1} \left(1 + \frac{3\gamma_1 \cdot k}{2n}\right)^{2(t+t_1)} \\
&\leqslant \left(1 - \frac{6\gamma_1 \cdot k}{n}\right)^{2t_1} \left(1 + \frac{3\gamma_1 \cdot k}{2n}\right)^{2(t+t_1)} \\
&\leqslant \left(1 - \frac{4\gamma_1 \cdot k}{n}\right)^{2t_1} \left(1 + \frac{3\gamma_1 \cdot k}{2n}\right)^{2t}
\end{aligned}$$

On the other hand, by Lemma 4.8.8,

$$\frac{\det_+(D^{s'})}{\det_+(D^{s^0})} \geqslant \left(1 - \frac{\gamma_1 k}{n}\right)^{2(t+t_1)}.$$

These two together imply that $t_1 \leqslant 2t$. □

B.5. Missing proofs from Section 4.9

Proof of Proposition 4.9.22. Multiplying the RHS out gives

$$\begin{pmatrix} L_{FF} & L_{FF}L_{FF}^{\dagger}L_{FC} \\ L_{CF}L_{FF}^{\dagger}L_{FF} & L_{CF}L_{FF}^{\dagger}L_{FF}L_{FF}^{\dagger}L_{FC} + L_{CC} - L_{CF}L_{FF}^{\dagger}L_{FC} \end{pmatrix}. \tag{B.12}$$

Then it suffices to show that $L_{CF}L_{FF}^{\dagger}L_{FF} = L_{CF}$, which will imply that (B.12) is equal to L. Notice that L_{CF} contains the edges between C and F. Moreover, we have $L_{FF} = L_{G[F]} + \sum_{(u,v)\in E\cap(F\times C)}(e_{u\leftarrow v})_F(e_{u\leftarrow v})_F^T$, and thus all rows of L_{CF} are in the range of L_{FF}, which gives that $L_{CF}L_{FF}^{\dagger}L_{FF} = L_{CF}$. □

Proof of Claim 4.9.31. It suffices to prove that $D_G^{\dagger/2}L_{G^2}D_G^{\dagger/2} \preceq 2D_G^{\dagger/2}L_GD_G^{\dagger/2}$. Notice that

$$D_G^{\dagger/2}L_GD_G^{\dagger/2} = I - D_G^{\dagger/2}A_GD_G^{\dagger/2}$$

and

$$D_G^{\dagger/2}L_{G^2}D_G^{\dagger/2} = I - D_G^{\dagger/2}A_GD_G^{\dagger}A_GD_G^{\dagger/2} = I - \left(D_G^{\dagger/2}A_GD_G^{\dagger/2}\right)^2.$$

Let $\lambda_1, \ldots, \lambda_n$ be the eigenvalues of $D_G^{\dagger/2}L_GD_G^{\dagger/2}$, and let $f_1, \ldots, f_n$ be a set of orthonormal eigenvectors. Then we have

$$\begin{aligned} D_G^{\dagger/2}L_GD_G^{\dagger/2} &= \sum_{i=1}^{n}\lambda_i f_i f_i^T \\ D_G^{\dagger/2}L_{G^2}D_G^{\dagger/2} &= \sum_{i=1}^{n}\left(1-(1-\lambda_i)^2\right)f_i f_i^T = \sum_{i=1}^{n}(2-\lambda_i)\lambda_i f_i f_i^T. \end{aligned}$$

Since $\lambda_i \in [0,2]$, we have our desired result. $\square$

Proof of Proposition 4.9.32. Notice that by definition

$$\begin{aligned}\sum_{u\in V} \mathrm{SC}_n(L, V\setminus\{u\}) =& \sum_{u\in V}\left(L_G - L_G(:,u)L_{uu}^{\dagger}L_G(:,u)^T\right)\\ =& nL_G - L_G D^{\dagger} L_G\\ =& n(D-A) - (D-A)D^{\dagger}(D-A)\\ =& nD - nA - (D - 2A + AD^{\dagger}A)\\ =& (n-2)(D-A) + D - AD^{\dagger}A = (n-2)L_G + L_{G^2}\end{aligned}$$

as desired. $\square$

Proof of Claim 4.9.33. It suffices to prove that

$$\mathbb{E}_{v_{i+1}}\left[(4.56) \mid V_i\right] = L_{H[V-V_i]}.$$

Let us calculate the LHS term by term.

$$\begin{aligned}&\mathbb{E}_{v_{i+1}}\left[L_{H[V-V_i]}(:,v_{i+1})\left(D_{v_{i+1}}^{H[V-V_i]}\right)^{\dagger} L_{H[V-V_i]}(:,v_{i+1})^T \mid V_i\right]\\ =&\frac{1}{t-i} L_{H[V-V_i]} D_{H[V-V_i]}^{\dagger} L_{H[V-V_i]}\\ =&\frac{1}{t-i}\left(D_{H[V-V_i]} - 2A_{H[V-V_i]} + A_{H[V-V_i]}D_{H[V-V_i]}^{\dagger}A_{H[V-V_i]}\right)\end{aligned}$$

$$\mathbb{E}_{v_{i+1}}\left[\left(1+\frac{2}{t-2-i}\right)L_{H[V-V_{i+1}]} \mid V_i\right] = \frac{t-2-i}{t-i}L_{H[V-V_i]} + \frac{2}{t-i}L_{H[V-V_i]} = L_{H[V-V_i]}$$

$$
\begin{aligned}
&\mathbb{E}_{v_{i+1}}\left[\frac{1}{t-i}\left(L_{H[V-V_i]^2}-2L_{H[V-V_i]}\right)\mid V_i\right]\\
=&\frac{1}{t-i}\left(L_{H[V-V_i]^2}-2L_{H[V-V_i]}\right)\\
=&\frac{1}{t-i}\left(D_{H[V-V_i]}-A_{H[V-V_i]}D^{\dagger}_{H[V-V_i]}A_{H[V-V_i]}\right)-\frac{2}{t-i}L_{H[V-V_i]}.
\end{aligned}
$$

One can verify that these three add up to $L_{H[V-V_i]}$. □

APPENDIX C

MISSING PROOFS IN CHAPTER 5

C.1. PRAM Primitives.

C.1.1. Eulerian Tours and Subtree Sum

Consider a tree T, and for each undirected edge (u, v) create two anti-parallel arcs (u, v) and (v, u); call this new graph T'. Then, T' admits an Eulerian tour, which can also be viewed as DFS traversal of T. This Eulerian tour can be used to compute a number of properties of T efficiently in parallel, and in this section we describe several of use to us. Most of these results are folklore and can be found in several textbooks on parallel algorithms, such as [Akl97].

Theorem C.1.1 (Eulerian Tour [Akl97]). *There is a $O(1)$ depth, $O(n)$ total work PRAM algorithm which outputs an ordered list of nodes corresponding to an Eulerian tour of a tree T with n nodes starting at a root r.*

Eulerian tours are usually computed via a linked list representation, where each directed arc points to the next element in the tour. This can also be converted to a random access ordered array, which is useful in several results below.

It is also known that one can compute all prefix sums of an array in $O(\log n)$ depth.

Theorem C.1.2 (Prefix Sums [Akl97]). *Given a list of values $a_1, \ldots, a_n$, there is an $O(\log n)$ depth, $O(n)$ work PRAM algorithm which outputs a new list $b_1, \ldots, b_n$ such that $b_k = \sum_{i \leqslant k} a_i$*

Using prefix sums and Eulerian tours, we can compute several useful properties of nodes in a tree.

Theorem C.1.3 ([Akl97]). *Let T be a tree rooted at r. Then, there is a PRAM algorithm which in $O(\log n)$ depth and $O(n)$ total work computes*

1. *The first and last time each node is visited by an Eulerian tour starting at r*
2. *The parent of each node*
3. *The depth of each node*

whenever there are at least $\Omega(n)$ processors.

Proof. Note that the length of the Eulerian tour array is always $O(n)$, as each original edge is traversed exactly twice and all trees over n nodes have $n-1$ edges. To find the first time each node is visited by a tour starting at r, assign one processor to each pair of consecutive elements in the list. Let u, v be these elements, appearing at indices $i-1, i$. The processor assigned to the pair then sets $b[(u,v)] = i$, where b is a new lookup table for the ordering of edges. To determine the first time a node u is visited by the tour, it suffices to compute $\min_v b[(v,u)]$, and similarly the last time can be determined by $\max_v b[(u,v)]$. These minimums and maximums can then be computed in depth $O(\log n)$ [Akl97]. The parent of any node first visited at position i in the list is the $i-1$ element in the Eulerian tour list.

For the depth each node, we say (u,v) is a "forward" arc if u is the parent of v, and a "reverse" edge if v is the parent of u. In the ordered list of arcs traversed by the Eulerian tour, assign each forward arc a value of $+1$ and each reverse arc a value of -1. Then, run prefix sum. Any cycle in the tour must have zero sum, since it must use both copies of each undirected edge it visits, and so the prefix sum whenever the tour vists v is the depth of v. Using the previously computed first time each node is visited, we may in parallel output the depth of every node. □

One of the most useful results of this this section is that we can compute *sumtree sums.*

Theorem C.1.4 (Subtree Sum)**.** *Let T be a tree where each node v is associated with a weight w_v. Then, there is a PRAM algorithm with $O(\log n)$ depth and $O(n)$ total work that computes, for every node u, the sum of the w_v in the subtree rooted at u.*

Proof. As in the proof of Theorem C.1.3, classify each arc in the Eulerian tour of T as either forward or reverse. Set the weight of each reverse arc to be 0, and the weight of each forward arc (u,v) to w_v. Then, run prefix sum on the Eulerian tour list of arcs. At the last occurrence of u, all arcs in the subtree of u must have already been traversed (since the Eulerian tour uses each arc exactly once, and does not visit u again), so the prefix sums includes the sum of all weights in this subtree. Moreover, between the first and last visit to u, the tour cannot leave the subtree of u; if it did, then one of the arcs between u and its parent must have been used at least twice. So, the sum of the subtree is exactly $w_u + p[u^{(-1)}] - p[u^{(1)}]$, where p is the array of prefix sums and $u^{(-1)}, u^{(1)}$ are the

indices of the last and first occurrences of u in the tour, respectively. The prefix sum along with first and last occurrences of each node can be computed in $O(\log n)$ depth and $O(n)$ work, and the final sum requires only $O(1)$ depth and $O(n)$ work. □

For example, subtree sums can be used to find the LCA of two nodes in a tree

Theorem C.1.5. *Let T be a tree rooted at r. Then, for any $u, v \in T$, there is an $O(\log n)$ depth, $O(n)$ work PRAM algorithm to find the least-common ancestor of u and v.*

Proof. The algorithm first sets weights of 0 on all nodes besides u and v, and sets weight 1 on u and v before computing subtree sums. Using the algorithm of Theorem C.1.3, also compute the depth of each node. Then, output the lowest-depth (i.e. furthest from the root) node who has subtree sum 2.

Computing the subtree sum and depth of each node can be done in $O(\log n)$ work and $O(n)$ work. Identifying all nodes with sum 2 can be done in $O(1)$ depth and $O(n)$ work, and taking the max depth of these can be done in $O(\log n)$ depth and $O(n)$ work, so the entire algorithm has the desired runtime.

For correctness, the algorithm computes the lowest-depth node whose subtree contains both u and v, which is exactly the LCA of u and v. □

C.1.2. Tree Separators

A tree separator node of a tree is a node whose removal results is a forest with components of at most half the size of the original tree.

Definition C.1.6 (Tree Separator Node, Definition 5.2.7)**.** A node q in a tree T is called a tree separator node of T if the forest induced by $T \setminus \{q\}$ consists of trees with at most $|T|/2$ nodes.

A very useful folklore results shows that every tree contains at least one tree separator node.

Lemma C.1.7 (folklore)**.** *For all trees T, there exists a node $q \in T$ such that q is a tree separator node of T.*

Proof. Suppose for contradiction T does not contain a tree separator node. Define M_u for each node $u \in T$ to be the size of the largest component of $T \setminus \{u\}$, and let $u' = \arg\min_{u \in T} M_u$. By

assumption, $M_{u'} > |T|/2$, and let C' be the component of $T \setminus \{u'\}$ with size $M_{u'}$. Since T is a tree, there is a unique neighbor of u' in C; call this node v. $|C'| > |T|/2$, so $|T \setminus C'| \leqslant |C'| + 1$ and $|T \setminus (C' \cup \{u'\})| \leqslant |C'|$. Thus, $C' \setminus \{v\}$ must be the largest component in $T \setminus \{v\}$, as the sum of all other components can be at most $|C'| - 1$. But then we have $M_v < M_{u'}$, contradicting that $M_{u'}$ is minimal. □

We now give an algorithm to find a tree separator node with logarithmic depth and linear work, and uses the subtree sum algorithm of Theorem C.1.4.

Lemma C.1.8. *There is a $O(\log k)$ depth, $O(k)$ work PRAM algorithm to find a tree separator node of any tree with k nodes.*

Proof. Let T be a tree with k nodes, and arbitrarily root T. The algorithm first assigns every node weight 1, and uses the algorithm of Theorem C.1.4 to compute subtree sums. Let s_u be the value of this subtree sum at node u. Then, in parallel, for each $u \in T$, check if $s_v \leqslant k/2$ for all v children of u, and check that $k - s_u \leqslant k/2$. Output any such u as a tree separator node.

Removing any tree separator node, by definition, results in a forest with components of size at most $k/2$, which is the exact condition the algorithm checks. A tree separator node exists for all trees (Lemma C.1.7), so the algorithm always outputs a tree separator node. The depth of the algorithm comes from the fact that the subtree sums and examining all nodes can be done in $O(\log k)$ depth. For work, note that the value of every node u needs to be checked twice: once when determining if $k - s_u \leqslant k/2$, and once when its (unique) parent checks if $s_u \leqslant k/2$. So, combined with the $O(k)$ work for subtree sums, the total work is $O(k)$. □

C.2. Parallel Implementation of the Cut-Matching Game of [RST14]

In this section, we discuss the implementation of `partition-A1` in Section 5.3.1, which in turn is built on top of the cut-matching game in [RST14]. In the remainder of this section, note that we always operate over the subdivision graph as defined in Definition 2.5.6, i.e., we have $V' = V \cup X_E$ where X_E is the set of split vertices of the edges in E, $E' = \bigcup_{e=(u,v)\in E} \{(u, x_e), (x_e, v)\}$, and $c'_{(u,x_e)} = c'_{(v,x_e)} = c_e$ for every edge $e \in E$. Moreover, for ease of exposition, whenever we refer

to a set of edges Y, we in fact refer to the subdivision vertices X_Y corresponding to set Y. It is straightforward to see that constructing the subdivision graph only takes $O(m)$ work and $O(1)$ depth.

All the algorithms that are presented in this section are identical to that of [RST14]; we merely discuss them for the sake of completeness and showing that they admit a $O(m \operatorname{polylog} n)$ work and $O(\operatorname{polylog} n)$ depth PRAM implementation. This entire process requires a blackbox access to only two subroutines: a $(1-\varepsilon)$-approximate max-flow algorithm $\mathcal{F}_\varepsilon$ that also returns the $(1+\varepsilon)$-approximate minimum cut, and a $(1-\delta)$-approximate flow-path decomposition algorithm $\mathcal{D}_\delta$ (whose implementation is given in Section 5.5). We shall henceforth refer to the work and depth of these algorithms $\mathcal{A} \in \{\mathcal{F}_\varepsilon, \mathcal{D}_\delta\}$ on an m-edge graph as $T(\mathcal{A}, m)$, and $D(\mathcal{A}, m)$. To build intuition, we begin by discussing the implementation specifics for the uncapacitated case in Section C.2.1 through Section C.2.4, and extend our result to the general capacitated case in Section C.2.5.

Using a cut-matching game to find well-linked edges. Before discussing [RST14], we shall find it useful to first understand the cut-matching game framework of [KRV06]. It is a technical tool connecting the problem of approximating sparsest cuts or alternatively, certifying expansion to max flows, and is vital in the aforementioned result of [RST14]. Conceptually, it is an alternating game between a cut-player and a matching-player, where the former produces a bisection of the vertices, and the latter responds by producing a perfect matching across this bisection that does not necessarily belong to the underlying graph, but can be embedded in it[31]. The game ends when either the cut player produces a bisection for which the matching player cannot find a perfect matching that can be embedded in it (i.e. a sparse cut has been found), or the union of the perfect matchings produced thus far form an expander (i.e. an expander can be embedded in the underlying graph, certifying its expansion). The objective of the cut player is to terminate this game as quickly as possible, whereas the matching player seeks to delay this. In their work, [KRV06] showed that there is a near-linear time strategy for the cut-player that guarantees that this game terminates in $O(\log^2 n)$ rounds, regardless of the matching player's strategy, and it is easiest to understand when

[31] Roughly speaking, we say a matching can be embedded into the graph, if there exist flow paths connecting every pair of matched nodes such that the net flow does not exceed the capacity of any edge.

viewed as the following *multicommodity flow problem*: initially, every vertex starts off with a unit of its own unique commodity (which we can represent as a n-dimensional one-hot encoded vector, which we shall henceforth refer to as a *flow vector* held by the vertex), and the goal is to uniformly distribute these commodities across all vertices with low congestion. In each round of the game, every matched pair of vertices (according to the perfect matching produced by the matching player) mix (average) their currently held commodities with each other through the matching edge. Since the perfect matching could be embedded into the underlying graph, this operation is guaranteed to be feasible. Now supposing in some round, every vertex ended up with a (near) uniform spread over all unique commodities, then we are done; it implies that the union of the perfect matchings must induce an expander, which moreover can be embedded into the underlying graph. A good cut strategy therefore must find bisections that mix these commodities quickly, and the key to arguing its existence is a potential function that measures the total distance of each vertex's currently held commodities from uniform. In particular, one can show that there always exists a bipartition that guarantees a multiplicative $(1 - \Omega(1/\log n))$ factor reduction in this potential, regardless of the perfect matching produced by the matching-player, guaranteeing that the game terminates in just $O(\log^2 n)$ rounds. Moreover, this bisection can be found efficiently. While this sketches the *analysis*, this procedure would be far too inefficient to implement as it would require explicitly maintaining the vector of currently held commodities for every vertex (which would naively take $O(n^2)$ work). Fortunately, [KRV06] showed that it suffices to consider a *sketch* that is a *random projection* of these flow vectors in every round and still obtain the same convergence guarantees with high probability. The matching players strategy is essentially a max-flow computation with the sources and sinks being the two sides of the bipartition. If the max flow value is $n/2$, then we can obtain a perfect matching via a flow-path decomposition. Otherwise, a perfect matching cannot be embedded into this cut.

The above cut-matching game provides the basis of the algorithm of [RST14] for finding well linked edges; intuitively, a set F of edges is well-linked if we can embed an *expander* between the subdivision vertices of these edges, and the cut-matching game is precisely the technical tool used to verify this. The algorithm of [RST14] starts off with a candidate set of edges F, and the cut-matching game

is used to either certify well-linkedness of this set F, or to produce a witness of non-expansion (a highly congested cut), in which case F is updated to a new set F_{new} (intuitively, to the edges in the most congested cut). However, in the event that F is updated, the entire game would have to restart, which would be too expensive (unless the updated set is a constant factor smaller than the old candidate set, in which case it is okay because the number of restarts would be bounded by $O(\log n)$). The modified game of [RST14] circumvents this issue by *reusing* rounds of the old game in the event that the candidate set F does not change by much. This is achieved by *moving* flow vectors from the old candidate set F to the new set F_{new} along paths of constant congestion (found by the flow decomposition algorithm $\mathcal{D}$). However, not all edges in the new set F_{new} may end up receiving a flow vector from the old set F. Following the notation of [RST14], we shall therefore denote all candidate sets $F := A \uplus R$, where A refers to the set of *active* edges that have a flow vector, and R refers to the remaining edges in F. The exact details of this matching player are discussed in Section C.2.3. The cut-player is also slightly different from the one in [KRV06], where instead of a strict bipartition, the player produces a disjoint edge sets corresponding to the "source" side and "sink" side satisfying certain necessary properties. The exact details of this cut player are discussed in Section C.2.2.

C.2.1. Computing the projections of flow vectors

We first discuss how to efficiently compute the *sketch*, i.e. random projection of current state of the flow vectors held by every active edge (which evolves starting with a one-hot encoded vector, mixing and moving in every round depending on the matching player's strategy in that round). We remind the reader that given a candidate set of active edges A, there are $|A|$ flow vectors, each in $\mathbb{R}^m$, and the naive way to average and move flow vector would take $O(|A| \cdot m) = O(m^2)$ work. However, we note that in the algorithmic process, the flow vectors are only used for cut player to find the sources and sinks, and the procedure is implemented by a *projection* onto a random unit vector which we show can be simulated in nearly-linear work. We give a subroutine for such an implementation.

Lemma C.2.1. *Let $\boldsymbol{r}$ be any unit vector of dimension m, and let $\mathcal{M}_t = M_t M_{t-1} \cdots M_1$ be a $m \times m$ matrix such that each M_i is supported over at most m' non-zero coordinates, and let $\{\boldsymbol{v}_i\}_{i=1}^m$ be m standard basis vectors that represent indicators for each of the m unique commodities. Then,*

there exists a PRAM algorithm that computes $\mu_i^{(t)} = \boldsymbol{r}^T \mathcal{M}_t \boldsymbol{v}_i$ *for every* i*, in* $O(t \log n)$ *depth and* $O((m + m')t)$ *work.*

Proof. We perform the multiplication from left to right: $\mu_i^{(t)} = ((\boldsymbol{r}^T M_t)M_{t-1})M_{t-2} \ldots$, where each time we multiply a vector with a matrix of at most m' nonzeros, which takes $O(\log n)$ depth and $O(m')$ work. Summing over t iterations gives the desired result. □

As we will see shortly, the operations on the flow vectors in the t^{th} round of the cut-matching game can all be simulated by matrix multiplication of $\boldsymbol{r}^T \mathcal{M}_t \boldsymbol{v}_i$ as prescribed in Lemma C.2.1. Moreover, in all applications where the above subroutine is invoked, we always have the sparsity property $m' = O(m \ \text{polylog}\, n)$ for each matrix M_i since they correspond to fractional matrices that mix or move flows produced by the flow decomposition algorithm $\mathcal{D}$, and $t = O(\text{polylog}\, n)$ due to the convergence guarantees of the cut-matching game in [RST14]. We will refer to these matrices M_i's as mix-or-move matrices. We remark that the idea was implemented by both [KRV06] and [RST14], albeit it was less explicitly stated.

C.2.2. Parallel implementation of the cut player

We now describe the parallel implementation of a single iteration of the cut player that achieves the guarantees in Lemma 3.3 in [RST14]. Recall that the cut player's strategy given a set of active edges A, is to find two different subsets $A_S \subseteq A$ and $A_T \subseteq A$ whose commodities are not *well-mixed*. This is done in the following way: for every edge $e \in A$, let $\mathbf{f}_e$ be the flow vector currently held by e. In round t of the game, this is precisely given by $\mathbf{f}_e = M_t M_{t-1} \ldots M_1 \boldsymbol{v}_e$, where $\boldsymbol{v}_e$ is the indicator (one-hot encoded vector) of edge e's unique commodity, and each M_i represents the fractional mix-or-move matrix representing the matching player's strategy (i.e. mixing or moving) in round i. Let $\boldsymbol{\mu} := (1/|A|) \sum_{e \in A} \mathbf{f}_e$ represent the average flow vector in that round. Given these quantities (neither explicitly computed), and a random vector $\boldsymbol{r}$, for every edge $e \in A$, let $\mu_e = \langle \mathbf{f}_e \,,\, r \rangle$, and $\bar{\mu} := \langle \boldsymbol{\mu} \,,\, \boldsymbol{r} \rangle$ denote the projection of the flow vector of edge e, and the average flow vector onto the random vector, respectively. For any $B \subseteq A$, the potential P_B is defined as

$$P_B := \sum_{f \in B} (\mu_f - \bar{\mu})^2 \,.$$

Now given a set of active edges A, the cut player precisely seeks to identify a set of source edges A_S, a set of sink edges A_T, along with a value η such that (rf. Lemma 3.3 in [RST14])

1. η separates the sets, i.e. $\max_{e \in A_S} \mu_e \leqslant \eta \leqslant \min_{f \in A_T} \mu_f$ or $\min_{e \in A_S} \mu_e \geqslant \eta \geqslant \max_{f \in A_T} \mu_f$,
2. $|A_S| \leqslant |A|/8$ and $|A_T| \geqslant |A|/2$
3. for every source edge $e \in A_S$, $|\mu_e - \eta|^2 \geqslant (1/9)|\mu_e - \bar{\mu}|^2$,
4. $\sum_{e \in A_S} |\mu_e - \bar{\mu}|^2 \geqslant (1/160) \sum_{e \in A} |\mu_e - \bar{\mu}|^2$,

Note that, unlike [KRV06], A_S and A_T does not need to be an exact bi-partition of A. The following procedure describes implementation of the cut player's strategy that meets the above requirements in detail.

Subroutine `find-sources-and-sinks` [RST14]

Input: subdivision graph $G' = (V', E')$; a set of active edges $A \subseteq E$; a sequence of t mix-or-move matrices $M_t, \ldots, M_1$ each of dimension $m \times m$ and supported over at most $O(m \operatorname{polylog} n)$ non-zero coordinates.

Output: a set of source edges $A_S \subset A$, a set of sink edges $A_T \subset A$, and a separation value η.

Procedure:

1. Sample a unit vector $\boldsymbol{r}$ uniformly at random, and project the flow vector of each $e \in A$ to $\boldsymbol{r}$ to get $\mu_e = \langle \mathcal{M}_t \boldsymbol{v}_e \,,\, \boldsymbol{r} \rangle$, where $\mathcal{M}_t = M_t M_{t-1} \ldots M_1$. This step is executed by initializing length-m flow vectors with $\boldsymbol{v}_e$ the indicator vector of e, and then running the algorithm from Lemma C.2.1.
2. Assuming w.l.o.g that $|\{e \in A \mid \mu_e < \bar{\mu}\}| \leqslant |\{e \in A \mid \mu_e \geqslant \bar{\mu}\}|$, pick $L = \{e \in A \mid \mu_e < \bar{\mu}\}$, and let $R = A \setminus L$.
3. Compute P_L and P_R. If $P_L \geqslant \frac{1}{20} P_A$, set A_S as the $|A|/8$ edges (or all edges in L if there are fewer than $|A|/8$ edges in L) with the smallest μ_e values from L, A_T as R, and η as $\bar{\mu}$.
4. Otherwise:
   (a) Let $\ell = \sum_{e \in L} |\mu_e - \bar{\mu}|$, and let $\eta = \bar{\mu} + 4\ell/|A|$.
   (b) Let A_T be the edges whose μ_e is at most η.
   (c) Construct new sets $R' = \{e \in A \mid \mu_e \geqslant \bar{\mu} + 6\ell/|A|\}$, and let A_S be the $|A|/8$ edges with the largest μ_e values from R'.

This subroutine is used repeatedly during the cut-matching game in the hierarchical decomposition procedure of [RST14]. We now show this subroutine can be implemented efficiently in the PRAM setting.

Claim C.2.2. *There is a PRAM implementation of the subroutine* `find-source-and-sinks` *that, given an arbitrary sequence of t matrices, each with support over $O(m \operatorname{polylog} n)$ nonzero entries, has $O(t \log n)$ depth and $O(tm \operatorname{polylog} n)$ work.*

Proof. By Lemma C.2.1, computing the projections μ_e's takes $O(\log n)$ depth and $O(m \operatorname{polylog} n)$

work since the support size of each mix-or-move matrix M_i is $O(m \operatorname{polylog} n)$. The steps that compute the average and the potential all take $O(\log n)$ depth and $O(m \operatorname{polylog} n)$ work. Finally, taking a subset from A (resp. L and R) by checking the μ values also takes $O(\log n)$ depth and $O(m)$ work. Summarizing the above steps gives the desired $O(\log n)$ depth and $O(m \operatorname{polylog} n)$ work. □

C.2.3. Parallel implementation of the matching player

We now discuss the parallel implementation of the matching player. The matching player takes as input the source edges A_S and the sink edges A_T computed by the cut player, and computes a partial fractional matching M between the subdivision nodes X_{A_S} and X_{A_T} in the subdivision graph G'. The matching player uses a $(1-\varepsilon)$-approximate max-flow algorithm $\mathcal{F}_\varepsilon$ [32] to ensure that the matching can be routed in G' with constant congestion, and uses the $(1-\delta)$-approximate flow-path decomposition algorithm $\mathcal{D}_\delta$ to construct a matching from the output of the flow algorithm. At this point, we remind the reader that the cut-matching game in [RST14] deviates from that of [KRV06]. In [RST14], the matching player chooses to perform either a *matching* step or a *deletion* step and the choice is made by flipping a fair random coin. The matching step is similar to [KRV06] where the flow vectors of the matched edges are *mixed* resulting in a reduction in potential. The deletion step deletes the flow vectors on some edges (potentially *moving* them to new edges) resulting in a new set of active edges which again leads to potential reduction. The following procedure describes the matching player's strategy from [RST14] in detail.

[32] We assume that $\mathcal{F}_\varepsilon$ is such that it also outputs an approximate min-cut. Note that Sherman's algorithm [She13b] satisfies this assumption.

Subroutine `match-or-delete` [RST14]

Inputs: subdivision graph $G' = (V', E')$; a set of candidate edges $F \subseteq E$, active edges $A \subseteq E$, remaining edges $R \subseteq E$; source edges $A_S \subset A$; sink edges $A_T \subset A$; set of edges B.

Output: a $m \times m$ mix-or-move matrix M with support size at most $O(m \text{ polylog } n)$; new candidate edges F_{new}; new active edges A_{new}; new remaining edges R_{new}; set B_{new}.

Procedure:

1. Construct a capacitated graph G'_{st}: add a super-source s and connect s to all subdivision vertices $x \in X_{A_S}$ with capacity 1; add a super-sink t and connect t to all subdivision vertices $x \in X_{A_T}$ with capacity $\frac{1}{2}$; add capacity 2 to all edges in G' that are not incident on s or t.
2. Run the flow algorithm $\mathcal{F}_\varepsilon$ on G'_{st} with $\varepsilon = \frac{1}{3\log^3 n}$ to obtain a flow f; let $C' \in E'$ be the set of edges that correspond to the approximate min *cut*, and let $C \in E$ be the corresponding set of cut edges in G.
3. Run the flow-decomposition algorithm $\mathcal{D}_\delta$ on f with $\delta = \frac{1}{3\log^3 n}$, and let G be the data-structure encoding the flow paths (see Lemma 5.5.2). For every edge (s, x) where $x \in X_{A_S}$ with flow $f_{(s,x)} \geqslant 1/2$, rescale the flow such that $f_{(s,x)} = 1$. For every flow path p that uses edge (s, x), rescale the flow on p by $1/f_{(s,x)}$ to make the flow path consistent, which can be done by propagating this rescaling top-down in G. Adjust capacities in G'_{st} to make this new flow feasible.
4. With probability $\frac{1}{2}$, enter the **matching case**:
   (a) Let M_0 be the matrix of the fractional matching between X_{A_S}, X_{A_T} induced at the top-level of G. If M_0 is a partial matching, then make it perfect by adding self loops. Then, the mix-or-move matrix is given by $M \leftarrow \frac{1}{2}M_0 + \frac{1}{2}I$ where I is the identity matrix.
   (b) **return** mix-or-move matrix M, candidate edges $F_{\text{new}} = F = A \uplus R$, set $B_{\text{new}} = B$
5. With probability $\frac{1}{2}$, enter the **deletion case**:
   (a) If $((A \cup R) \setminus A_T) \cup C$ induces a balanced clustering in G, **return** new candidate edges $F_{\textbf{new}} = ((A \cup R) \setminus A_T) \cup C$. In this case, $|F_{\text{new}}| \leqslant (7/8)|F|$ due to which the cut-matching game restarts from scratch, and the remaining return values are not useful.
   (b) Else set $F_{\textbf{new}} = ((A \cup R) \setminus A_S) \cup C$: Let M_0 be the fractional matching between X_{A_S}, X_{A_T} induced at the top level of G. For each matched pair $(x, x') \in (X_{A_S} \times X_{A_T})$ in M_0, identify the first edge y in C on the flow path $x \rightsquigarrow x'$ using the data-structure G and construct a mix-or-move matrix M which moves the flow vector (instead of mixing it) from x to y. If the total flow received by y from all $x \in A_S$ exceeds 1, rescale this total flow to have value 1. Otherwise, zero out this flow, and add this edge y to set C_B.
   (c) Set $A_{\text{new}} = (A \setminus A_S) \cup (C \setminus C_B)$[a], $R_{\text{new}} = F_{\text{new}} \setminus A_{\text{new}}$.
   (d) **return** mix-or-move matrix M, F_{new}, A_{new}, R_{new}, $B_{\text{new}} = B \cup C_B$.

[a] Note that this update is mentioned as $A_{\text{new}} = (A \setminus A_T) \cup (C \setminus C_B)$ in [RST14] which is a typo.

Note that although the flow decomposition is an approximation version, the total flow value induced by the decomposition is at least $(1 - \frac{1}{3\log^3 n})^2 \geqslant (1 - \frac{1}{\log^3 n})$ of the max-flow value (assuming $n \geqslant 6$), which satisfies the desired requirement as in [RST14]. We now show that the matching-or-deletion subroutine can be implemented under the PRAM setting efficiently.

Claim C.2.3. *There is a PRAM implementation of the subroutine* `match-or-delete` *using depth* $O\left(D(\mathcal{F}_\varepsilon, m) + D(\mathcal{D}_\delta, m) + \operatorname{polylog} n\right)$ *and work* $O\big(T(\mathcal{F}_\varepsilon, m) + T(\mathcal{D}_\delta, m) + m \operatorname{polylog} n\big)$ *in total, where* $D(\mathcal{A}, m)$ *and* $T(\mathcal{A}, m)$ *are the depth and work required by the algorithm* $\mathcal{A} \in \{\mathcal{F}_\varepsilon, \mathcal{D}_\delta\}$ *for* $\varepsilon, \delta = 1/(3\log^3 n)$*, respectively.*

Proof. The first step in the algorithm is the construction of the graph G'_{st} which can be done in $O(1)$ depth and $O(m)$ work. The set C' is returned by $\mathcal{F}_\varepsilon$, and C can be constructed from C' in $O(1)$ depth and $O(m \operatorname{polylog} n)$ work. There are at most $O(m \operatorname{polylog} n)$ flow paths. Hence, using the data structure G returned by the flow decomposition algorithm $\mathcal{D}_\delta$, re-scaling of the flows and adjusting capacities takes $O(\log n)$ depth and $O(m \operatorname{polylog} n)$ work. We now analyze the required depth and work for both the matching and the deletion cases:

- In the matching case, since the fractional matching matrix has support size $O(m \operatorname{polylog} n)$, adding self-loops and computing the mixing matrix M takes $O(m \operatorname{polylog} n)$ work and $O(1)$ depth.
- In the deletion case, checking whether an induced clustering is balanced in Line 5a can be done by simply deleting the candidate edges and checking the sizes of the resulting connected components which takes $O(m)$ work and $O(\log n)$ depth. If we enter Line 5b, since there are at most $O(m \operatorname{polylog} n)$ matched pairs (x, x'), identifying the first edge in C on the flow path $x \rightsquigarrow x'$ for every matched pair requires $O(\operatorname{polylog}(n))$ depth and $O(m \operatorname{polylog}(n))$ work using the data structure G. The moving matrix M can then be computed in $O(m \operatorname{polylog} n)$ work and $O(1)$ depth. Rescaling and deleting flows can also be performed using $O(m \operatorname{polylog} n)$ work and $O(1)$ depth.

Taking the worst-case work and depth among the cases gives us the desired statement. □

C.2.4. Putting it together: `partition-A1`

In this section we combine the implementation of the cut and matching players to achieve the guarantees of Lemma 3.1 in [RST14]. This is also the algorithm that achieves the guarantees we claim in Lemma 5.3.3. Specifically, given a set of edges F that induces a 3/4-balanced clustering of the graph, the goal of Lemma 3.1 is to find a new set of edges F_{new} such that (rf. Lemma 3.1 in [RST14])

1. either $|F_{\text{new}}| \leqslant \frac{7}{8}|F|$;
2. or $F_{\text{new}} = A \cup R$ such that $|A| \leqslant |F|$, $|R| \leqslant \frac{2}{\log n} A$, and the edges in A are $\Omega(1/\log^2 n)$-well-linked.

The following presents the details of the algorithm that is used to prove Lemma 3.1 in [RST14].

Subroutine `partition-A1`

Input: Subdivision graph $G' = (V', E')$ of $G = (V, E)$; a set of edges $F \subseteq E$ that induce a 3/4-balanced partition of V.

Output: A new set of edges F_{new} that also induces a 3/4-balanced partition of V such that either

1. $|F_{\text{new}}| \leqslant (7/8)|F|$, or
2. $F_{\text{new}} = A \cup R$ with A, R disjoint such that edges in A are $\Omega(1/\log^2 n)$-well-linked, and $|R| \leqslant 2|A|/\log n$.

Procedure (cut-matching game):

1. Initialize matching matrix M_0 corresponding to self-loops; $A = F$; $R = \emptyset$; $B = \emptyset$.
2. For $t = 1, 2, \ldots, O(\log^2 n)$:
   (a) $(A_S, A_T, \eta) \leftarrow$ `find-source-and-sinks`$(G', A, \{M_i\}_{0 \leqslant i < t})$.
   (b) $(M_t, F_{\text{new}}, A_{\text{new}}, R_{\text{new}}, B_{\text{new}}) \leftarrow$`match-or-delete`$(G', F, A, R, A_S, A_T, B)$.
   (c) If $|F_{\text{new}}| \leqslant (7/8)|F|$: **return** F_{new} (return condition 1).
   (d) Else, update $B = B_{\text{new}}$, $A = A_{\text{new}}$, $R = R_{\text{new}}$.
   (e) Compute potential P_A of active edges A.
   (f) If $P_A \leqslant 1/(16n^2)$ (flow vectors in A have mixed):
      i. If $|B| \leqslant 2|A|/\log n$: **return** $F_{\text{new}} = A \cup B$ (return condition 2).
      ii. Else, we necessarily have $|A \cup R| \leqslant (7/8)|F|$: **return** $F_{\text{new}} = A \cup R$ (return condition 1).

With Claims C.2.2 and C.2.3, it is not difficult to check that the whole procedure of step 2 can be implemented in poly-logarithmic depth and nearly-linear work. We formalize this as follows.

Lemma C.2.4. *There is a PRAM implementation of the above algorithm that has depth* $O\left((D(\mathcal{F}_\varepsilon, m) + D(\mathcal{D}_\delta, m)) \cdot \operatorname{polylog} n\right)$ *and work* $O\big((m + T(\mathcal{F}_\varepsilon, m) + T(\mathcal{D}_\delta, m)) \cdot \operatorname{polylog} n\big)$ *for* $\varepsilon, \delta = O(1/\log^3 n)$.

Proof. Firstly, the number of iterations of the cut-matching game is upper bounded by $O(\log^2 n)$. Then the lemma follows immediately from the work and depth calculations of the cut and matching

steps in Claims C.2.2 and C.2.3, respectively. □

C.2.5. Extending to the capacitated case

Although [RST14] states that their results hold for capacitated graphs, they do not detail this extension. However, for completeness, we sketch the implementation of the cut matching game in [RST14] on capacitated graphs, and discuss how they can be implemented in PRAM. Throughout this section we assume graphs have integer capacities bounded by $\text{poly}(n)$.

We note that the analysis of the cut matching game follows by viewing each edge e of capacity c_e as c_e uncapacitated parallel copies. It thus remains to show that the cut matching game can be implemented with $O(m \operatorname{polylog} n)$ work and $\operatorname{polylog} n$ depth, assuming a $(1-\varepsilon)$-approximate maxflow algorithm with the same work and depth. We first describe a key subroutine that we need throughout the cut matching game, namely *averaging of flow vectors*, and then describe the changes we make to the cut and matching players.

Averaging flow vectors. Note that naively, treating each capacitated edge as uncapacitated copies can potentially lead to a large number of edges, and maintaining flow vectors on them will be too costly. Therefore we will always maintain the invariant that the parallel copies of the same capacitated edge carries the same flow vector. Whenever this variant gets violated (e.g. after a flow mixing or deletion step), we restore it by averaging out the flow vectors across the parallel copies. Thereby, we never store more than $O(m \operatorname{polylog}(n))$ flow vectors at any point. Notice that in the actual implementation of the cut matching game, we never explicitly store the flow vectors but only store their projections; and thus we only need to average the projections of these vectors, which is equivalent to first averaging the flow vectors and then taking projections by linearity. The averaging of the projections can be done in $O(\log n)$ depth and near-linear work by parallel summation.

Moreover, as we show in the claim below, the potential function only decreases after the averaging of any collection of flow vectors. Recall that in a graph with m_0 uncapacitated edges $e_1, \ldots, e_{m_0}$ with flow vectors $f_{e_1}, \ldots, f_{e_{m_0}}$ on them, the potential function in [RST14] is defined to be

$$\Phi_f := \min_c \sum_{i=1}^{m_0} \|f_{e_i} - c\|^2 = \sum_{i=1}^{m_0} \|f_{e_i} - \mu\|^2 ,$$

where $\mu = \frac{1}{m_0}\sum_{i=1}^{m_0} f_{e_i}$ is the average of all flow vectors.

Claim C.2.5. *Let $E' \subseteq E$ be any collection of uncapacitated edges. Define new flow vectors $f'_{e_1}, \ldots, f'_{e_{m_0}}$ by averaging out the flow vectors on edges in E', namely,*

$$f'_e = \begin{cases} \frac{1}{|E'|}\sum_{e\in E'} f_e & e \in E' \\ f_e & e \notin E' \end{cases}.$$

Then the potential function can only decrease going from f to f':

$$\Phi_f \geqslant \Phi'_f.$$

Proof. Let $\mu = \frac{1}{m_0}\sum_{i=1}^{m_0} f_{e_i}$ be the average flow vector of all edges. Note that this is also the average flow vector with respect to f', since averaging the flow vectors of edges in E' does not change the total sum of the flow vectors. Thus it suffices to compare the contribution of edges in E' to the potential function with respect to f, f' respectively. To this end, we write the contribution with respect to f' as

$$\begin{aligned}\sum_{e\in E'} \left\|f'_e - \mu\right\|^2 &= \sum_{e\in E'} \left\|\frac{1}{|E'|}\sum_{e\in E'} f_e - \mu\right\|^2 \\ &= |E'| \cdot \left\|\frac{1}{|E'|}\sum_{e\in E'} f_e - \mu\right\|^2.\end{aligned}$$

Letting $\mathcal{D}$ denote the uniform distribution over edges in E', we can write the above contribution as

$$|E'| \cdot \left\|\mathbb{E}_{e\in\mathcal{D}}\left[f_e\right] - \mu\right\|^2.$$

Notice that the function $f(x) = \|x - \mu\|^2$ is quadratic and hence also convex. Therefore we can

apply Jensen's inequality and obtain

$$\begin{aligned}|E'| \cdot \|\mathbb{E}_{e\in\mathcal{D}}[f_e] - \mu\|^2 \leqslant & |E'| \cdot \mathbb{E}_{e\in\mathcal{D}}\left[\|f_e - \mu\|^2\right] \\ = & \sum_{e\in E} \|f_e - \mu\|^2 .\end{aligned}$$

That is, the contribution of edges in E' with respect to f' is at most that with respect to f, implying that the potentail of f' can only be smaller than that of f, as desired. □

The cut and matching players. The cut player uses the exact same strategy to find the sources and sinks, except that now they do not explicitly operate all parallel copies, but rather manipulate the parallel copies of the same edge together, by exploiting the fact that they all have the same flow vectors. For example, when computing projection of the flow vectors, we only need to do the computation once for the parallel copies of the same edge; when computing the average of the flow vectors, we just need to compute a weighted average where each flow vector is weighted by the capacity of the corresponding edge. Notice that when choosing A_S, we could end up only choosing a subset of the parallel copies of the same edge, but leaving the remaining parallel copies out. Then the matching player will create a flow graph by connecting super source to the split vertex of the edge with capacity being the number of parallel copies in A_S.

The matching player also adopts almost the same strategy as the uncapacitated case, with the following modifications:

1. Most notably, in both matching case and deletion case, where we mix or move flow vectors, we average the flow vectors of parallel copies of the same edge afterwards. This is because different copies of the same edge could be matched differently, resulting in different flow vectors after mixing/moving. However, since the matching we found has support size $O(m\,\mathrm{polylog}(n))$, the total number of distinct flow vectors is always bounded by $O(m\,\mathrm{polylog}(n))$. Notice that once again, we never explicitly average the flow vectors but only average their projections.
2. At Line 3 of `match-or-delete`, after we find a flow decomposition, we scale the flow paths as follows. For every edge (s, x) with flow $f_{(s,x)} \geqslant (1/2)c_{(s,x)}$, rescale the flow such that

$f_{(s,x)} = c_{(s,x)}$; For every flow path p that uses edge (s, x), rescale the flow on p by $c_{(s,x)}/f_{(s,x)}$ to make the flow path consistent, which can be done by propagating this rescaling top-down in the flow decomposition DAG in $O(m \operatorname{polylog}(n))$ work and $\operatorname{polylog}(n)$ depth. Adjust capacities in G'_{st} accordingly.

3. At Line 5b of `match-or-delete`, if the total flow received by y exceeds c_y, rescale this total flow to have value c_y. Otherwise, zero out this flow, and add (the parallel copies of) this edge y to set C_B. Notice that the rescaling of the flow can again be done by propagating it top-down in the flow decomposition DAG in $O(m \operatorname{polylog}(n))$ work and $\operatorname{polylog}(n)$ depth.

C.3. Parallel Implementation of Sherman's Algorithm

In this section, we discuss the implementation details of vanilla Sherman's algorithm [She13b] in the PRAMmodel, which forms the basis of the near-linear work, polylogarithmic depth approximate max-flow subroutine invoked in our cutting-scheme. This algorithm consists of an outer-algorithm that makes $O(\log m)$ many iterative calls to an inner procedure `almost-route` that actually performs the gradient descent. At the end of these calls, we are left with a minimum congestion flow that is *almost* feasible, in the sense that there is a negligible residual demand that can be routed in the flow network with $O(1/\operatorname{poly}(m))$ congestion. This outer-algorithm terminates by routing this residual demand along a maximum spanning tree, achieving feasibility of the superimposed flows (i.e. the resultant flow routes the desired demands b). This relatively simpler outer algorithm is described below, with the bulk of the technical detail being contained in the `almost-route` subroutine that implements gradient descent.

It is easy to see that the above outer-algorithm admits an efficient PRAMimplementation, assuming that the subroutine `almost-route` admits a PRAMimplementation with near-linear work and polylogarithmic depth; there are only $O(\log n)$ many iterations in the outer algorithm, and lines 1,4,6, and 8 can easily be implemented with $O(m)$ work and $O(1)$ depth, and the maximum spanning tree construction in line 7 has a known $O(m)$ work, $O(\operatorname{polylog} n)$ depth PRAMalgorithm [PR02].

We next discuss the implementation specifics of the subroutine `almost-route` which actually performs the optimization and is more involved. The key idea behind this subroutine is to transform the

Algorithm C.3.1 `approximate-maximum-flow`$(G, R, b, \alpha, \varepsilon)$

Input: Graph $G = (V, E, c)$; α-congestion approximator R that is a hierarchical decomposition of G; vertex demands $b \in \mathbb{R}^V$; quality of the congestion approximator $\alpha = O(\text{polylog}\, n)$; precision $\varepsilon > 0$.

Output: A $(1+\varepsilon)$-approximate minimum congestion flow $f \in \mathbb{R}^E$ that routes demands b; $(1-\varepsilon)$-approximate maximum congested cut S.

Procedure:

1: $b_0 \leftarrow b$; compute B, the vertex-edge incidence matrix of G.
2: $(f_0, S_0) \leftarrow$ `almost-route`$(G, R, b_0, \alpha, \varepsilon)$
3: **for** $i \leftarrow 1 \ldots \log(2m)$ **do**
4: $\quad b_i \leftarrow b_{i-1} - Bf_{i-1}$
5: $\quad (f_i, S_i) \leftarrow$ `almost-route`$(G, R, b_i, \alpha, 1/2)$. ▷ subsequent S_i are not needed
6: Let $t = \log(2m)$ be the final iteration counter of the above loop; set $b_T \leftarrow b_t - Bf_t$.
7: Compute the maximum spanning tree T of G.
8: Let f_T be the flow obtained by routing demands b_T on the tree T
9: **return** flow $f = f_T + \sum_{i=1}^{\log(2m)} f_i$; cut $S = (S_0, \overline{S_0})$.

constrained optimization problem (for undirected graphs) given in Eqn. 1.1 into an unconstrained one using the following *congestion potential* function

$$\phi(f) = \mathsf{lmax}(C^{-1}f) + \mathsf{lmax}(2\alpha R(b - Bf)),$$

where for any $x \in \mathbb{R}^k$,

$$\mathsf{lmax}(x) := \log\left(\sum_{i=1}^{k}(e^{x_i} + e^{-x_i})\right)$$

is the symmetric softmax function, a differentiable approximation of $||\cdot||_\infty$. Algorithmically, this is achieved via a standard gradient descent which finds a $(1-\varepsilon)$-approximate maximum flow f after at most $O(\varepsilon^{-3}\alpha^2 \log n)$ iterations. Therefore, an efficient PRAMimplementation of this algorithm effectively reduces to finding an efficient implementation of a single iteration of the descent step within this algorithm, a formal description of which is given below.

Observe that implementing the above algorithm requires us to compute (i) the value of the potential function $\phi(f)$, and (ii) the partial derivatives $\partial\phi(f)/\partial f_e$ of the potential with respect to the flow on each edge in the graph. Additionally, we also need to bound the total number of iterations of the above algorithm (lines 2, 4), for which we directly leverage the result of Sherman

Algorithm C.3.2 `almost-route`$(G, R, b, \alpha, \varepsilon)$

Input: Graph $G = (V, E, c)$; α-congestion approximator R that is a hierarchical decomposition of G; vertex demands $b \in \mathbb{R}^V$; quality of the congestion approximator $\alpha = O(\text{polylog}\, n)$; precision $\varepsilon > 0$.

Output: A $(1+\varepsilon)$-approximate minimum congestion flow $f \in \mathbb{R}^E$ that routes demands b; $(1-\varepsilon)$-approximate maximum congested cut S.

Procedure:

1: Initialize $f \leftarrow 0$; compute $k_b \leftarrow (16 \log n)/(2\alpha\varepsilon||Rb||_\infty)$; scale $b \leftarrow k_b \cdot b$.
2: **repeat**
3: Set $k_f \leftarrow 1$; scaling factor $s \leftarrow 17/16$.
4: **while** $\phi(f) < 16\varepsilon^{-1} \log n$ **do**
5: Scale $k_f \leftarrow s \cdot k_f$; $f \leftarrow s \cdot f$; $b \leftarrow s \cdot b$.
6: Set $\delta \leftarrow \sum_{e \in E} \left| c_e \cdot \frac{\partial \phi(f)}{\partial f_e} \right|$.
7: **if** $\delta \geqslant \varepsilon/4$ **then**
8: For each edge $e \in E$, update $f_e \leftarrow f_e - \text{sign}\left(\frac{\partial \phi(f)}{\partial f_e}\right) \cdot \frac{\delta c_e}{1+4\alpha^2}$.
9: **else**
10: Undo scaling $f \leftarrow f/k_f$, $b \leftarrow b/(k_b k_f)$.
11: Compute the maximum congested cut $(S, \overline{S})$ from the $(n-1)$ threshold cuts of vertex potentials $\{\pi_v\}_{v \in V}$ induced due to $\partial\phi(f)/\partial f_e$ (described shortly).
12: **return** flow f, cut S .
13: **until termination**

(Lemma 2.5 in [She13b]), which shows that the total number of iterations until termination (line 2) is $O(\alpha^2 \varepsilon^{-3} \log n)$, and within each iteration, the total number of times we scale the flow and demands (line 4) is $O(\log \alpha)$. We now show how to compute the value of the potential, and its partial derivatives. In order to do so, we shall find it instructive to understand the structure of the congestion approximator R.

The congestion approximator $R \in \mathbb{R}^{x \times n}$ is a matrix with each row $i \in [x]$ corresponding to a cut in the graph, and each column corresponding to a vertex. For any cut $i = (S_i, \overline{S_i})$ considered by the congestion approximator, entry $R_{i,v} \in \{0, 1\}$ indicates whether vertex v lies on the S_i side of the cut, normalized by the total capacity $c(S_i, \overline{S_i})$ of the cut, i.e. the sum of capacities of all edges crossing this cut. Therefore, the product $[Rb']_i$ of this row of the congestion approximator with any demand vector b' gives the congestion that would be induced by routing these demands across the cut $(S_i, \overline{S_i})$. However, we cannot explicitly construct this matrix due to work and depth constraints, and instead shall use the specific structure of the congestion approximator to efficiently compute

these congestion values for all cuts explicitly considered by the approximator.

The congestion approximator in our case is a $O(\log n)$ depth rooted tree T corresponding to a hierarchical decomposition of the flow instance $G = (V, E)$ upon which Sherman's algorithm is invoked, with the leaves corresponding to the vertices $v \in V$ in the flow network, and the internal nodes corresponding to a cluster consisting of the leaf vertices in the subtree rooted at that internal node. This hierarchical decomposition tree T can equivalently be viewed as a set of cuts in the input graph; each node $i \in T$ in this tree corresponds to a cut $(S_i, \overline{S_i})$, where S_i is the set of vertices corresponding to the leaves in the subtree rooted at node i in the tree T. In the following analysis, we shall leverage this view of the congestion approximator in order to efficiently compute the value of the congestion potential, as well as its partial derivatives.

We begin by decomposing the congestion potential into its two components

$$\phi(f) = \phi_1(f) + \phi_2(f); \text{ where } \phi_1(f) = \mathsf{lmax}(C^{-1}f), \text{ and } \phi_2(f) = \mathsf{lmax}(2\alpha R(b - Bf)).$$

To compute the first component $\phi_1(f)$, we can simply compute the congestion f_e/c_e of every edge $e \in E$ in parallel, followed by an aggregation step, which can be done with $O(m)$ total work and $O(\log n)$ depth.

To compute the second component $\phi_2(f)$, we can compute the residual demands $b_v - \sum_{e \in \delta(v)} B_{v,e} f_e$ for every vertex $v \in V$ where $\delta(v)$ corresponds to the set of edges incident on vertex v. This also requires $O(m)$ total work and $O(\log n)$ depth (with every vertex first reading the flow values of incident incoming edges followed by those of incident outgoing edges in two separate passes to avoid read collisions). The total demand of any subset of vertices in the congestion approximator (rooted-tree) can then be computed with $\widetilde{O}(n)$ total work and $O(\log n)$ depth using subtree sums. Given the capacity $c(S_i, \overline{S_i})$ of every cut $i = (S_i, \overline{S_i})$ represented by the internal nodes in our congestion approximator, the second term can then be computed via an aggregation, which can be done with $O(m)$ total work and $O(\log n)$ depth.

To compute the partial derivatives, we first consider the component $\phi_1(f)$ in our congestion poten-

tial. Then we have that for any edge $e \in E$, the partial derivative

$$\frac{\partial \phi_1(f)}{\partial f_e} = \frac{\exp(f_e/c_e) - \exp(-f_e/c_e)}{c_e \cdot \exp(\phi_1(f))}$$

which can easily be computed with $O(m)$ work and $O(1)$ depth when the potential $\phi_1(f)$ is known (its computation is described above).

To compute the partial derivative of the second component $\phi_2(f)$, let $\mathcal{I}$ be the set of all cuts (rows) considered by our congestion approximator, and for any cut $i = (S_i, \overline{S_i}) \in \mathcal{I}$, let $y_i = 2\alpha[R(b - Bf)]_i$ be the congestion induced by the residual demands across cut $i = (S_i, \overline{S_i})$. Then we have for any edge $e \in E$, the partial derivative

$$\frac{\partial \phi_2(f)}{\partial f_e} = \sum_{i \in \mathcal{I}} \frac{\partial \phi_2(f)}{\partial y_i} \cdot \frac{\partial y_i}{\partial f_e} = \sum_{i \in \mathcal{I}} \frac{\exp(y_i) - \exp(-y_i)}{\exp(\phi_2(f))} \cdot \frac{2\alpha B_{S_i,e}}{c(S_i, \overline{S_i})},$$

where $c(S_i, \overline{S_i})$ is the capacity of the cut $i = (S_i, \overline{S_i})$ considered in our congestion approximator, and (with some abuse of notation) $B_{S_i,e} = \sum_{v \in S_i} B_{v,e} \in \{-1, 0, 1\}$ is an indicator of whether in cut i, edge e is an incoming edge (1), outgoing edge (-1) or does not cross it (0). The cuts $\mathcal{I}$ are not arbitrary. Rather, they are induced by a single rooted hierarchical decomposition tree T, which we can use to efficiently compute this partial derivative for every edge. For an edge $e = (u, v) \in E$, let $T_{u,v}$ be the unique path between u, v in T. Then we have that

$$\frac{\partial \phi_2(f)}{\partial f_e} = \sum_{i \in T_{u,v}} \frac{\exp(y_i) - \exp(-y_i)}{\exp(\phi_2(f))} \cdot \frac{2\alpha B_{S_i,e}}{c(S_i, \overline{S_i})}.$$

Now observe that for any internal node i (corresponding to some cut $(S_i, \overline{S_i})$) that is encountered on the path $T_{u,x}$ between u and the *least-common-ancestor* x of u, v in the rooted tree T, we have $B_{S_i,e} = -1$, and for any internal node i that is encountered on the path $T_{x,v}$ between v and x in $\mathcal{T}$, we have $B_{S_i,e} = +1$. Now for any internal node j in T, let $T_{j,r}$ denote the unique path in T from

the root r of T to node j. Then we can define for every internal node j in T, a node potential π_j as

$$\pi_j := \sum_{i \in T_{j,r}} \frac{\exp(y_i) - \exp(-y_i)}{\exp(\phi_2(f))} \cdot \frac{2\alpha}{c(S_i, \overline{S_i})},$$

which is easy to compute with $O(n)$ total work and $O(\log n)$ depth through a prefix sum on an Eulerian tour of T that starts and ends at the root r of T. This is achieved by setting the weight of the forward edge entering an internal node i from its parent to be $+(\exp(y_i) - \exp(-y_i))/(\exp(\phi_2(f))) \cdot (2\alpha/c(S_i, \overline{S_i}))$ and the reverse edge leaving the internal node i going to its parent to be $-(\exp(y_i) - \exp(-y_i))/(\exp(\phi_2(f))) \cdot (2\alpha/c(S_i, \overline{S_i}))$. Therefore, sum corresponding to subtrees in the prefix sums evaluate to 0, leaving just the sum of the root r to node i path. Given these node potentials, it is now easy to compute the partial derivatives of any edge $e = (u, v)$ as

$$\frac{\partial \phi_2(f)}{\partial f_e} = \pi_v - \pi_u,$$

which requires just $O(m)$ total work and $O(\log n)$ depth.

Lastly, Sherman shows that these vertex potentials π_v induced by the flow when it is approximately optimal (i.e. when the subroutine terminates) also allow us to efficiently recover the approximate minimum cut (equivalently, the approximate maximum congested cut). Specifically, one of the threshold cuts with respect to the vertex potentials is an approximate min-cut, and this can be computed efficiently in $\widetilde{O}(m)$ total work and $O(\log n)$ depth by sorting the vertices by their potential values and returning the most congested cut from the resulting $n - 1$ threshold cuts. Therefore, we have that Sherman's algorithm admits an efficient implementation in the PRAMmodel.

C.4. Computing Min Cut on Trees

To compute a hierarchical decomposition on trees in Section 5.2.2, we need to compute an *exact* s-t min-cut cut on a congestion approximator tree with the addition of a super-source s and super-sink t. In this section, we show how to compute the exact min-cut when the tree has $O(\log n)$ depth, which is simpler, before extending it to trees with arbitrary depth.

The easy case: $O(\log n)$ depth.

Let T be a tree rooted at r with $O(\log n)$ depth, let $s \notin T$ be a super-source, and $t \notin T$ be a super-sink; s and t may be connected arbitrarily to T and these edges may have arbitrary capacity. For a s-t min-cut $(S, \bar{S})$ of $T \cup \{s, t\}$, without loss of generality $s \in S$ and $t \in \bar{S}$. As such, it remains to determine which nodes of T are in S and which are in $\bar{S}$, and so we consider finding a s-t min-cut as a tree problem. To account for the capacities of edges incident on s or t, for each $u \in T$, we set node weights $w_u^s = c_{us}$ and $w_u^t = c_{ut}$, where c_{us}, c_{ut} are the capacities of the (u, s) and (u, t) edges, respectively, or 0 if no such edge exists.

To find the min-cut capacity, we can use dynamic programming. For each $u \in T$, define $\texttt{cut}^s(u)$ and $\texttt{cut}^t(u)$ as the s-t min-cut capacity of the subtree rooted at u with the restriction that u is on the s side of the cut (i.e. $u \in S$) or u is on the t side of the cut (i.e. $u \in \bar{S}$), respectively. The recurrence relations are then $\texttt{cut}^s(u) = w_u^t + \sum_{v \in D(u)} \min\{\texttt{cut}^s(v), c_{uv} + \texttt{cut}^t(v)\}$ and $\texttt{cut}^t(u) = w_u^s + \sum_{v \in D(u)} \min\{c_{uv} + \texttt{cut}^s(v), \texttt{cut}^t(v)\}$, where $D(u)$ is the set of children of u. Since we assume T has $O(\log n)$ depth, standard dynamic programming techniques allow us to compute $\min\{\texttt{cut}^s(r), \texttt{cut}^t(r)\}$, which is the s-t min-cut capacity, in $O(\log n)$ depth and $O(n)$ work. For brevity, we have presented computing only the capacity of the min-cut, but the actual cut may be found by storing the argmin for each minimum taken in the recurrence relations.

Before continuing to the arbitrary depth case, we introduce a (slightly) generalized problem where some vertices are constrained to be in S, or $\bar{S}$, an extension useful when extending to trees of arbitrary depth.

Definition C.4.1 ((F_s, F_t)-Restricted s-t Min-Cut)**.** Let T be a tree, let s be a super-source and let t be a super-sink. Then, given disjoint subsets F_s, F_t of nodes of T, a (F_s, F_t)-*Restricted* s-t *Min-Cut* of $T \cup \{s, t\}$ is a minimum s-t cut $(S, \bar{S})$ under the restriction that $F_s \cup \{s\} \subseteq S$ and $F_t \cup \{t\} \subseteq \bar{S}$.

The DP presented before can be easily modified to also solve this extended version when T has

$O(\log n)$ depth, using recurrence relations

$$\texttt{res\_cut}^s(u) = \begin{cases} w_u^t + \sum_{v \in D(u)} \min\{\texttt{res\_cut}^s(v), c_{uv} + \texttt{res\_cut}^t(v)\} & \text{if } u \notin F_t \\ \infty & \text{if } u \in F_t \end{cases} \tag{C.1}$$

$$\texttt{res\_cut}^t(u) = \begin{cases} w_u^s + \sum_{v \in D(u)} \min\{c_{uv} + \texttt{res\_cut}^s(v), \texttt{res\_cut}^t(v)\} & \text{if } u \notin F_s \\ \infty & \text{if } u \in F_s \end{cases} \tag{C.2}$$

where again $D(u)$ is the set of children of $u \in T$.

Lemma C.4.2. *Let T be a tree on n nodes rooted at r with depth $O(\log n)$, and let $\texttt{res\_cut}^s$ and $\texttt{res\_cut}^t$ be defined as in* (C.1) *and* (C.2)*. Then, given disjoint subsets F_s, F_t of vertices of T, $\min\{\texttt{res\_cut}^s(r), \texttt{res\_cut}^t(r)\}$ is the capacity of an (F_s, F_t)-Restricted s-t Min-Cut on $T \cup \{s, t\}$. Moreover, this value can be computed using an $O(\log n)$ depth and $O(n)$ total work PRAM algorithm.*

Proof. By straightforward dynamic programming, since T has $O(\log n)$ depth, $\texttt{res\_cut}^s(r)$ and $\texttt{res\_cut}^t(r)$ can be computed in $O(\log n)$ depth and $O(n)$ work. For correctness, first note that the base cases are correct: the cost of any solution where $x \in F_s$ is placed in $\bar{S}$ is infinite (and analogously for F_t and S) and, by construction, w_u^s is the capacity of the edge between u and s, if it exists (and similarly for w_u^t). The correctness then follows by induction. □

Extending to arbitrary depth trees.

When T has super-logarithmic depth, we modify the DP and divide into subproblems based on tree separator nodes (see Definition 5.2.7) rather than children. For a DP subproblem to find the s-t min-cut on a tree T', we compute a tree separator node q and recurse on each tree of $T' \setminus \{q\}$. From the definition of a tree separator node, this results in subproblems on trees which are at most half the size of T'.

When combining subproblems, we use the recursive calls to determine the min-cut capacity when $q \in S$ and the min-cut capacity with $q \in \bar{S}$, and return the lower value. To do this, we must also determine for each $u \in \delta(q)$ (i.e. each neighbor of q in T') whether $u \in S$ or $u \in \bar{S}$, in order to determine whether to add the capacity of the (u, q) edge to the cut. As such, for every neighbor u

of q, we compute 2 subproblems on the connected component of $T' \setminus \{q\}$ containing u: one where we constrain $u \in S$, and one where we constrain $u \in \bar{S}$. Due to recursion, in any given subproblem there might be multiple vertices which are constrained to be in S or $\bar{S}$; we call these the *tracked* vertices. Each subproblem then takes as input a tree T', a set of tracked vertices $\mathcal{A}$, and subset $\mathcal{S} \subseteq \mathcal{A}$, where the goal of the subproblem is to compute a s-t min-cut $(S, \bar{S})$ on T' under the restriction that $\mathcal{S} \subseteq S$ and $(\mathcal{A} \setminus \mathcal{S}) \subseteq \bar{S}$.

However, there are $2^{|\mathcal{A}|}$ possible subsets $\mathcal{S} \subseteq \mathcal{A}$, and so the number of possible subproblems (and thus total work) is exponential in the number of tracked vertices. As such, we must bound the number of tracked vertices in any subproblem. To do this, if we ever have a subproblem with 3 tracked vertices, rather than recursing on the components formed by removing a tree separator node, we recurse on the trees formed by removing the LCA of 2 tracked vertices. By rooting every subtree at a tracked vertex, this results in new subproblems with at most 2 tracked vertices, and so all subproblems have at most 3 tracked vertices. Importantly, these additional steps to reduce the number of tracked vertices at most double the depth of the recursion, leading to $O(\log n)$ levels of recursion. Since each level of recursion can be implemented in $O(\log n)$ depth and $O(n)$ work via dynamic programming, we obtain a $O(\log^2 n)$ depth, $\tilde{O}(n)$ work PRAM algorithm.

Below we present the full algorithm for general trees. For brevity, we present computing the capacity of the min-cut; the actual cut may be found by storing the argmin for each min taken in the recurrence relation. We reuse the notation from the algorithm for $O(\log n)$ depth, with S denoting the side of the cut containing s and w_u^s, w_u^t denoting the capacity of the (u, s) or (u, t) edge, respectively, and 0 if no such edge exists.

`min-cut-on-trees` **Subroutine**

Inputs: Tree T with root r, super-source $s \notin T$ and super-sink $t \notin T$, both connected arbitrarily to T.

Goal: For T' a subtree of T, $\mathcal{A} \subseteq T'$ a set of tracked vertices and $\mathcal{S} \subseteq \mathcal{A}$, recursively compute $\texttt{cut}(T', \mathcal{A}, \mathcal{S})$, which is the capacity of a s-t min-cut $(S, \bar{S})$ on $T' \cup \{s, t\}$ such that $\mathcal{S} \cup \{s\} \subseteq S$ and $(\mathcal{A} \setminus \mathcal{S}) \cup \{t\} \subseteq \bar{S}$.

Output: $\texttt{cut}(T, \emptyset, \emptyset)$ as the s-t min-cut capacity.

Computing $\texttt{cut}(T', \mathcal{A}, \mathcal{S})$**:**

If the depth of T' is at most $10 \log n$, run the algorithm of Lemma C.4.1 on $T' \cup \{s, t\}$, with $F_s = \mathcal{S}$ and $F_t = \mathcal{A} \setminus \mathcal{S}$ (and T' rooted arbitrarily).

Otherwise, with T' rooted at any tracked vertex when $|\mathcal{A}| \neq \emptyset$ and arbitrarily otherwise:

1. Compute a split vertex q:
   (a) If $|\mathcal{A}| \leqslant 2$, set q to be a tree separator node of T', using the algorithm of Lemma C.1.8.
   (b) If $|\mathcal{A}| = 3$, set q as the LCA of 2 non-root tracked vertices, using the algorithm of Theorem C.1.5.
2. Compute the connected components of $T' \setminus \{q\}$, and for each $u \in \delta(q)$[a] define C^u as the component containing u.
3. For each $u \in \delta(q)$, update tracked vertices $\mathcal{A}^u = (\mathcal{A} \cap C^u) \cup \{u\}$ and $\mathcal{S}^u = \mathcal{S} \cap \mathcal{A}^u$.
4. Compute the min-cut capacity conditioned on $q \in S$:
$$\texttt{cut}_{q \in S} = w_q^t + \sum_{u \in \delta(q)} \min\{w_u^t + \texttt{cut}(C^u, \mathcal{A}^u, \mathcal{S}^u \cup \{u\}),\ w_u^s + c_{uq} + \texttt{cut}(C^u, \mathcal{A}^u, \mathcal{S}^u)\}$$
5. Similarly, compute the min-cut capacity conditioned on $q \in \bar{S}$:
$$\texttt{cut}_{q \in \bar{S}} = w_q^s + \sum_{u \in \delta(q)} \min\{w_u^t + c_{uq} + \texttt{cut}(C^u, \mathcal{A}^u, \mathcal{S}^u \cup \{u\}),\ w_u^s + \texttt{cut}(C^u, \mathcal{A}^u, \mathcal{S}^u)\}$$
6. If $q \notin \mathcal{A}$, return
$$\texttt{cut}(T', \mathcal{A}, \mathcal{S}) = \min\{\texttt{cut}_{q \in S}, \texttt{cut}_{q \in \bar{S}}\}$$
7. If $q \in \mathcal{A}$ and $q \in \mathcal{S}$, return $\texttt{cut}(T', \mathcal{A}, \mathcal{S}) = \texttt{cut}_{q \in S}$.
8. If $q \in \mathcal{A}$ and $q \notin \mathcal{S}$, return $\texttt{cut}(T', \mathcal{A}, \mathcal{S}) = \texttt{cut}_{q \in \bar{S}}$.

[a] $\delta(q)$ is the set of neighbors of q in T'

Lemma C.4.3 (Min Cut on Trees)**.** *The algorithm* `min-cut-on-trees` *computes the s-t min-cut capacity of a tree T with the addition of a super-source s and super-sink t in $O(\log^2 n)$ depth and*

$\tilde{O}(n)$ *work.*

Proof. To bound the depth and work of the algorithm, we first show that the number of tracked vertices in any subproblem is always at most 3. The algorithm begins with a call that has no tracked vertices, and each recursive call adds at most one additional tracked vertex to each subproblem. So, to bound the number of tracked vertices, it suffices to show that if a subproblem has 3 tracked vertices, the number of tracked vertices in each recursive call is at most 2. Consider the call $\texttt{cut}(T', \mathcal{A}, \mathcal{S})$ with $|\mathcal{A}| = 3$, and let q be the LCA computed in Line 1. By the definition of an LCA and the fact that we always root T' at a tracked vertex, it follows that the path in T' between any 2 tracked vertices passes through q. As such, after removing q from T', each element of $\mathcal{A}$ is in a separate connected component. Thus, the tracked sets used in the recursive calls each have at most 2 elements: at most one element from $\mathcal{A}$, and one neighbor of q.

We claim there are at most $O(\log n)$ recursive levels of the algorithm, where recursive level i consists of all subproblems resulting from i consecutive recursive calls to $\texttt{cut}(T, \emptyset, \emptyset)$. Consider a subproblem to compute $\texttt{cut}(T', \mathcal{A}, \mathcal{S})$ with at most 2 tracked vertices (i.e. $|\mathcal{A}| \leqslant 2$). In this case, we recurse on subtrees which are connected components after the removal of a tree separator node. By the definition of a tree separator node (Definition 5.2.7), this removal results in subtrees which are at most half the size of T'. So, there can be at most $O(\log n)$ such steps before the recursion terminates. Now, suppose $|\mathcal{A}| = 3$ and the subproblem $\texttt{cut}(T', \mathcal{A}, \mathcal{S})$ occurs at level i. The resulting subproblems in level $i + 1$ have at most 2 tracked vertices, and so in level $i + 2$, the size of the resulting subtrees is at most half the size of T'. Thus, there are $O(\log n)$ total levels. Each recursive level can be processed in parallel, in $O(\log n)$ depth (to compute connected components, subtree sums, and combining recursive calls), making the total depth of the algorithm $O(\log^2 n)$.

The subtrees (i.e. all distinct T' from subproblems) at each level of recursion form a partition of the nodes of T. Moreover, by rooting trees and computing the split vertex q deterministically, every subproblem on T' has the same set of tracked vertices. So, since there are at most 3 tracked vertices in any subproblem, there are at most 8 possible sets $\mathcal{S} \subseteq \mathcal{A}$, and thus at most 8 subproblems using any one subtree. The work to compute a k node subtree, outside of recursion, is at most $O(k)$. So, the total work at each level is $O(n)$, and as there are $O(\log n)$ levels, the complete work is $\tilde{O}(n)$.

For subtrees with depth $O(\log n)$, the correctness follows from Lemma C.4.1. For trees with larger depth, we must have either $q \in S$ or $q \in \bar{S}$, where q is the split vertex computed in Line 1. By induction and the setting of the edge weights w_u^s, w_u^t, $\texttt{cut}_{q \in S}$ is the min-cut with the restrictions on $\mathcal{A}$ imposed by $\mathcal{S}$ when $q \in S$ (and similarly for $\texttt{cut}_{q \in \bar{S}}$). So, as we take the min of $\texttt{cut}_{q \in S}, \texttt{cut}_{q \in \bar{S}}$ when $q \notin \mathcal{A}$, $\texttt{cut}(T', \mathcal{A}, \mathcal{S})$ is the correct min-cut capacity. □

C.5. Ensuring Polynomial Aspect Ratio

In this section, we give a $O(\log n)$ depth, $\tilde{O}(m)$ work PRAM algorithm which, given s, t and ε converts any capacitated graph G into one with with $\text{poly}(n/\varepsilon)$ aspect ratio that preserves the s-t maximum flow up to a $(1 - \varepsilon)$ factor.

Algorithm C.5.1 `lower-aspect-ratio`(G, s, t, ε)

Input: G: graph with arbitrary capacities.
s, t: source and sink in $V(G)$.
ε: error parameter.
Output: G': graph with $\text{poly}(n/\varepsilon)$ aspect ratio.
Procedure:

1: Initialize $G' \leftarrow G$.
2: Let T be a maximum spanning tree of G, computed using the algorithm of [PR02].
3: Compute c' as the capacity of the lowest capacity edge on the unique s-t path in T.
4: For any edge e in G' with capacity larger than mc', reduce its capacity to mc'.
5: For any edge e in G' with capacity less than $\varepsilon c'/m$, delete e.
6: **return** G'

The algorithm of [PR02] has $O(\log n)$ depth and $O(m)$ work and finding all edges on the s-t path in T can be done in $O(\log n)$ depth and $\tilde{O}(m)$ work using subtree sums, so computing c' and modifying the capacities can be done in $O(\log n)$ depth and $\tilde{O}(m)$ work. Moreover, by construction, the modified graph G' has aspect ratio $m^2/\varepsilon = \text{poly}(n/\varepsilon)$ and no capacities have been increased, so it remains to show that the maximum s-t flow does not reduce by more than a $(1 - \varepsilon)$ factor.

For this, we first need the following simple lemma about maximum spanning trees.

Lemma C.5.1. *Let G be a capacitated graph and let T be a maximum capacity spanning tree of G. Let P be the unique s-t path in T (which is also a path in G), and let P' be any other s-t path in G. Then,*

$$\min_{e \in P} c_e \geqslant \min_{e \in P'} c_e$$

where c_e is the capacity of edge e.

Proof. Let $c' = \min_{e \in P} c_e$, and suppose for contradiction there exists in G a s-t path P' such that for all $e \in P'$, $c_e > c'$. Let e' be an edge on P with capacity c'. Removing e' from T results in exactly two connected components S and $V \setminus S$ (with $s \in S$ and $t \in V \setminus S$). Since P' is an s-t path, it must contain an edge $h = (u, v)$ such that $u \in S$ and $v \in V \setminus S$, so removing e' from T while adding h results in a spanning tree T'. But $c_h > c_{e'}$ by assumption, and so the capacity of T' is greater than that of T, contradicting that T is a maximum spanning tree of G. □

This then allows us to show the procedure decreases the maximum flow by at most a ε factor.

Lemma C.5.2. *Let G be a capacitated graph and let $G' =$ `lower-aspect-ratio`(G, s, t, ε). Suppose it possible to route f units of flow from s to t in G. Then, it is possible to route $(1 - \varepsilon)f$ units of flow from s to t in G'.*

Proof. Let T be a maximum spanning tree of G, and let c' be the capacity of the minimum capacity edge on the unique s-t path in T. By Lemma C.5.1, all other s-t paths in G have minimum capacity at most c'; thus, every s-t path can support at most c' units of flow. There can be at most m disjoint paths from s to t, and so it follows that $f \leqslant c'm$. Thus, reducing the capacity of all edges with capacity greater than mc' to mc' does not affect the maximum flow. Similarly, clearly $f \geqslant c'$, as there is a s-t path with minimum capacity c'. Thus, as the sum flow on edges of capacity at most $\varepsilon c'/m$ is at most $\varepsilon c'$, deleting edges of capacity at most $\varepsilon c'/m$ reduces the flow by at most εf. It then follows that the max flow in G' must have value at least $(1 - \varepsilon)f$. □

BIBLIOGRAPHY

[AAV20] Noga Alon, Yossi Azar, and Danny Vainstein. Hierarchical clustering: A 0.585 revenue approximation. In Jacob D. Abernethy and Shivani Agarwal, editors, *Conference on Learning Theory, COLT 2020, 9-12 July 2020, Virtual Event [Graz, Austria]*, volume 125 of *Proceedings of Machine Learning Research*, pages 153–162. PMLR, 2020.

[AB19] Sepehr Assadi and Aaron Bernstein. Towards a unified theory of sparsification for matching problems. In *2nd Symposium on Simplicity in Algorithms, SOSA@SODA 2019, January 8-9, 2019 - San Diego, CA, USA*, pages 11:1–11:20, 2019.

[AB21] Sepehr Assadi and Soheil Behnezhad. Beating two-thirds for random-order streaming matching. In *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPIcs*, pages 19:1–19:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[ABB+19] Sepehr Assadi, MohammadHossein Bateni, Aaron Bernstein, Vahab S. Mirrokni, and Cliff Stein. Coresets meet EDCS: algorithms for matching and vertex cover on massive graphs. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 1616–1635, 2019.

[ABKL23] Sepehr Assadi, Soheil Behnezhad, Sanjeev Khanna, and Huan Li. On regularity lemma and barriers in streaming and dynamic matching. In Barna Saha and Rocco A. Servedio, editors, *Proceedings of the 55th Annual ACM Symposium on Theory of Computing, STOC 2023, Orlando, FL, USA, June 20-23, 2023*, pages 131–144. ACM, 2023.

[ACC+18] Moab Arar, Shiri Chechik, Sarel Cohen, Cliff Stein, and David Wajc. Dynamic Matching: Reducing Integral Algorithms to Approximately-Maximal Fractional Algorithms.

In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*, pages 7:1–7:16, 2018.

[ACL+22] Sepehr Assadi, Vaggos Chatziafratis, Jakub Lacki, Vahab Mirrokni, and Chen Wang. Hierarchical clustering in graph streams: Single-pass algorithms and space lower bounds. In Po-Ling Loh and Maxim Raginsky, editors, *Conference on Learning Theory, 2-5 July 2022, London, UK*, volume 178 of *Proceedings of Machine Learning Research*, pages 4643–4702. PMLR, 2022.

[ADL+92] Noga Alon, Richard A. Duke, Hanno Lefmann, Vojtech Rödl, and Raphael Yuster. The algorithmic aspects of the regularity lemma (extended abstract). In *33rd Annual Symposium on Foundations of Computer Science, Pittsburgh, Pennsylvania, USA, 24-27 October 1992*, pages 473–481. IEEE Computer Society, 1992.

[AGM12a] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Analyzing graph structure via linear measurements. In Yuval Rabani, editor, *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 459–467. SIAM, 2012.

[AGM12b] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Graph sketches: sparsification, spanners, and subgraphs. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 5–14, 2012.

[Akl97] S.G. Akl. *Parallel Computation: Models and Methods*. Prentice Hall, 1997.

[AKL+24] Arpit Agarwal, Sanjeev Khanna, Huan Li, Prathamesh Patil, Chen Wang, Nathan White, and Peilin Zhong. Parallel approximate maximum flows in near-linear work and polylogarithmic depth. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 3997–4061. SIAM, 2024.

[AKLP22a] Arpit Agarwal, Sanjeev Khanna, Huan Li, and Prathamesh Patil. Sublinear algo-

rithms for hierarchical clustering. *Advances in Neural Information Processing Systems*, 35:3417–3430, 2022.

[AKLP22b] Arpit Agarwal, Sanjeev Khanna, Huan Li, and Prathamesh Patil. Sublinear algorithms for hierarchical clustering. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2022 (NeuIPS 2022)*, 2022.

[Alo02] Noga Alon. Testing subgraphs in large graphs. *Random Struct. Algorithms*, 21(3-4):359–370, 2002.

[AM85] Noga Alon and Vitali D Milman. $\lambda 1$, isoperimetric inequalities for graphs, and superconcentrators. *Journal of Combinatorial Theory, Series B*, 38(1):73–88, 1985.

[ANOY14] Alexandr Andoni, Aleksandar Nikolov, Krzysztof Onak, and Grigory Yaroslavtsev. Parallel algorithms for geometric graph problems. In David B. Shmoys, editor, *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 574–583. ACM, 2014.

[AO95] Ravindra K Ahuja and James B Orlin. A capacity scaling algorithm for the constrained maximum flow problem. *Networks*, 25(2):89–98, 1995.

[ARV09a] Sanjeev Arora, Satish Rao, and Umesh Vazirani. Expander flows, geometric embeddings and graph partitioning. *Journal of the ACM (JACM)*, 56(2):1–37, 2009.

[ARV09b] Sanjeev Arora, Satish Rao, and Umesh V. Vazirani. Expander flows, geometric embeddings and graph partitioning. *J. ACM*, 56(2):5:1–5:37, 2009.

[Ass22] Sepehr Assadi. A two-pass (conditional) lower bound for semi-streaming maximum matching. In *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms, SODA 2022, Virtual Conference / Alexandria, VA, USA, January 9 - 12, 2022*, pages 708–742. SIAM, 2022.

[ASZ20] Alexandr Andoni, Clifford Stein, and Peilin Zhong. Parallel approximate undirected shortest paths via low hop emulators. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 322–335, 2020.

[AW21] Josh Alman and Virginia Vassilevska Williams. A refined laser method and faster matrix multiplication. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 522–539. SIAM, 2021.

[BDL21] Aaron Bernstein, Aditi Dudeja, and Zachary Langley. A Framework for Dynamic Matching in Weighted Graphs. In *Proccedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2021, to appear*, 2021.

[Ber06] Pavel Berkhin. A survey of clustering data mining techniques. In Jacob Kogan, Charles K. Nicholas, and Marc Teboulle, editors, *Grouping Multidimensional Data - Recent Advances in Clustering*, pages 25–71. Springer, 2006.

[Ber20] Aaron Bernstein. Improved bounds for matching in random-order streams. In *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*, pages 12:1–12:13, 2020.

[BHI18] Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. Deterministic Fully Dynamic Data Structures for Vertex Cover and Matching. *SIAM J. Comput.*, 47(3):859–887, 2018.

[BHN16] Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. New Deterministic Approximation Algorithms for Fully Dynamic Matching. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 398–411. ACM, 2016.

[BK96] András A. Benczúr and David R. Karger. Approximating s-t minimum cuts in $\tilde{O}(n^2)$ time. In Gary L. Miller, editor, *Proceedings of the Twenty-Eighth Annual ACM Sympo-*

sium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996, pages 47–55. ACM, 1996.

[BK04] Yuri Boykov and Vladimir Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE transactions on pattern analysis and machine intelligence*, 26(9):1124–1137, 2004.

[BK15] András A. Benczúr and David R. Karger. Randomized approximation schemes for cuts and flows in capacitated graphs. *SIAM J. Comput.*, 44(2):290–319, 2015.

[BK22] Soheil Behnezhad and Sanjeev Khanna. New trade-offs for fully dynamic matching via hierarchical EDCS. In *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms, SODA 2022, Virtual Conference / Alexandria, VA, USA, January 9 - 12, 2022*, pages 3529–3566. SIAM, 2022.

[BKS17] Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. *J. ACM*, 64(6), oct 2017.

[BM99] Andrej Brodnik and J. Ian Munro. Membership in constant time and almost-minimum space. *SIAM J. Comput.*, 28(5):1627–1640, 1999.

[BS15] Aaron Bernstein and Cliff Stein. Fully dynamic matching in bipartite graphs. In Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann, editors, *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part I*, volume 9134 of *Lecture Notes in Computer Science*, pages 167–179. Springer, 2015.

[BS16] Aaron Bernstein and Cliff Stein. Faster fully dynamic matchings with small approximation ratios. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 692–711. SIAM, 2016.

[CC17] Moses Charikar and Vaggos Chatziafratis. Approximate hierarchical clustering via sparsest cut and spreading metrics. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 841–854. SIAM, 2017.

[CCL+15] Dehua Cheng, Yu Cheng, Yan Liu, Richard Peng, and Shang-Hua Teng. Efficient sampling for gaussian graphical models via spectral sparsification. In *Proceedings of The 28th Conference on Learning Theory, COLT 2015, Paris, France, July 3-6, 2015*, pages 364–390, 2015.

[CCN19] Moses Charikar, Vaggos Chatziafratis, and Rad Niazadeh. Hierarchical clustering better than average-linkage. In Timothy M. Chan, editor, *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 2291–2304. SIAM, 2019.

[CF14] Graham Cormode and Donatella Firmani. A unifying framework for ℓ_0-sampling algorithms. *Distributed Parallel Databases*, 32(3):315–335, 2014.

[CH09] Bala G Chandran and Dorit S Hochbaum. A computational study of the pseudoflow and push-relabel algorithms for the maximum flow problem. *Operations research*, 57(2):358–376, 2009.

[CKL+22a] Li Chen, Rasmus Kyng, Yang P Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 612–623. IEEE, 2022.

[CKL22b] Yu Chen, Sanjeev Khanna, and Huan Li. On weighted graph sparsification by linear sketching. In *63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2022, Denver, CO, USA, October 31 - November 3, 2022*, pages 474–485. IEEE, 2022.

[CKM+14] Michael B. Cohen, Rasmus Kyng, Gary L. Miller, Jakub W. Pachocki, Richard Peng, Anup B. Rao, and Shen Chen Xu. Solving SDD linear systems in nearly $m\log^{1/2}n$ time. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 343–352, 2014.

[CKM17] Vincent Cohen-Addad, Varun Kanade, and Frederik Mallmann-Trenn. Hierarchical clustering beyond the worst-case. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 6201–6209, 2017.

[CKMM18] Vincent Cohen-Addad, Varun Kanade, Frederik Mallmann-Trenn, and Claire Mathieu. Hierarchical clustering: Objective functions and algorithms. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 378–397. SIAM, 2018.

[CKMM19] Vincent Cohen-Addad, Varun Kanade, Frederik Mallmann-Trenn, and Claire Mathieu. Hierarchical clustering: Objective functions and algorithms. *J. ACM*, 66(4):26:1–26:42, 2019.

[CLM+15] Michael B. Cohen, Yin Tat Lee, Cameron Musco, Christopher Musco, Richard Peng, and Aaron Sidford. Uniform sampling for matrix approximation. In *Proceedings of the 2015 Conference on Innovations in Theoretical Computer Science, ITCS 2015, Rehovot, Israel, January 11-13, 2015*, pages 181–190, 2015.

[CNC18] Vaggos Chatziafratis, Rad Niazadeh, and Moses Charikar. Hierarchical clustering with structural constraints. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan,*

Stockholm, Sweden, July 10-15, 2018, volume 80 of *Proceedings of Machine Learning Research*, pages 773–782. PMLR, 2018.

[Coh95] Edith Cohen. Approximate max-flow on small depth networks. *SIAM Journal on Computing*, 24(3):579–597, 1995.

[CS19] Yi-Jun Chang and Thatchaphol Saranurak. Improved distributed expander decomposition and nearly optimal triangle enumeration. In Peter Robinson and Faith Ellen, editors, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 66–73. ACM, 2019.

[CSWZ16] Jiecao Chen, He Sun, David P. Woodruff, and Qin Zhang. Communication-optimal distributed clustering. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 3720–3728, 2016.

[Dan51] George B Dantzig. Application of the simplex method to a transportation problem. *Activity analysis and production and allocation*, 1951.

[Das16] Sanjoy Dasgupta. A cost function for similarity-based hierarchical clustering. In *Proceedings of the Forty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '16, pages 118–127. Association for Computing Machinery, 2016.

[Din70] EA Dinic. Algorithm for solution of a problem of maximum flow in a network with power estimation, soviet math. doll. 11 (5), 1277-1280,(1970). *English translation by RF. Rinehart*, 1970.

[Din73] EA Dinic. Metod porazryadnogo sokrashcheniya nevyazok i transportnye zadachi (excess scaling and transportation problems). *Issledovaniya po Diskretnoi Matematike*, 1973.

[DKP+17] David Durfee, Rasmus Kyng, John Peebles, Anup B. Rao, and Sushant Sachdeva. Sampling random spanning trees faster than matrix multiplication. In Hamed Hatami, Pierre McKenzie, and Valerie King, editors, *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 730–742. ACM, 2017.

[DMR18] Luc Devroye, Abbas Mehrabian, and Tommy Reddad. The total variation distance between high-dimensional gaussians. *arXiv preprint arXiv:1810.08693*, 2018.

[EK72] Jack Edmonds and Richard M Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM (JACM)*, 19(2):248–264, 1972.

[ER+60] Paul Erdos, Alfréd Rényi, et al. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5(1):17–60, 1960.

[ESBB98] Michael B Eisen, Paul T Spellman, Patrick O Brown, and David Botstein. Cluster analysis and display of genome-wide expression patterns. *Proceedings of the National Academy of Sciences*, 95(25):14863–14868, 1998.

[ET75] Shimon Even and R Endre Tarjan. Network flow and testing graph connectivity. *SIAM journal on computing*, 4(4):507–518, 1975.

[FF56] Lester Randolph Ford and Delbert R Fulkerson. Maximal flow through a network. *Canadian journal of Mathematics*, 8:399–404, 1956.

[FGL+21] Sebastian Forster, Gramoz Goranci, Yang P. Liu, Richard Peng, Xiaorui Sun, and Mingquan Ye. Minor sparsifiers and the distributed laplacian paradigm. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*, pages 989–999. IEEE, 2021.

[FHHP19] Wai-Shing Fung, Ramesh Hariharan, Nicholas J. A. Harvey, and Debmalya Panigrahi.

A general framework for graph sparsification. *SIAM Journal on Computing*, 48(4):1196–1223, 2019.

[FHM+20] Alireza Farhadi, Mohammad Taghi Hajiaghayi, Tung Mai, Anup Rao, and Ryan A. Rossi. Approximate maximum matching in random streams. In *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 1773–1785, 2020.

[FKM+05] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. *Theor. Comput. Sci.*, 348(2-3):207–216, 2005.

[FKN21] Arnold Filtser, Michael Kapralov, and Navid Nouri. Graph spanners by sketching in dynamic streams and the simultaneous communication model. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 1894–1913, 2021.

[FKS89] Joel Friedman, Jeff Kahn, and Endre Szemeredi. On the second eigenvalue of random regular graphs. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 587–598, 1989.

[Fox11] Jacob Fox. A new proof of the graph removal lemma. *Annals of Mathematics*, 174(1):561–579, 2011.

[Fri01] Alan M Frieze. Edge-disjoint paths in expander graphs. *SIAM Journal on Computing*, 30(6):1790–1801, 2001.

[FS21] Moran Feldman and Ariel Szarf. Maximum matching sans maximal matching: A new approach for finding maximum matchings in the data stream model. *CoRR*, abs/2109.05946. To appear in APPROX 2022., 2021.

[FW78] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Pro-*

ceedings of the tenth annual ACM symposium on Theory of computing, pages 114–118, 1978.

[GKK12] Ashish Goel, Michael Kapralov, and Sanjeev Khanna. On the communication and streaming complexity of maximum bipartite matching. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 468–485. SIAM, 2012.

[Gol08] Andrew V. Goldberg. The partial augment–relabel algorithm for the maximum flow problem. In *Algorithms - ESA 2008*, pages 466–477. Springer Berlin Heidelberg, 2008.

[Gol17] Oded Goldreich. *Introduction to Property Testing*. Cambridge University Press, 2017.

[GR98] Andrew V Goldberg and Satish Rao. Beyond the flow decomposition barrier. *Journal of the ACM (JACM)*, 45(5):783–797, 1998.

[GR99] Oded Goldreich and Dana Ron. A sublinear bipartiteness tester for bounded degree graphs. *Comb.*, 19(3):335–373, 1999.

[GRST21] Gramoz Goranci, Harald Räcke, Thatchaphol Saranurak, and Zihan Tan. The expander hierarchy and its applications to dynamic graph algorithms. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2212–2228. SIAM, 2021.

[GSZ+11a] Frédéric Gilbert, Paolo Simonetto, Faraz Zaidi, Fabien Jourdan, and Romain Bourqui. Communities and hierarchical structures in dynamic social networks: analysis and visualization. *Social Network Analysis and Mining*, 1(2):83–95, 2011.

[GSZ11b] Michael T. Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the mapreduce framework. In *Algorithms and Computation - 22nd International Symposium, ISAAC 2011, Yokohama, Japan, December 5-8, 2011. Proceedings*, volume 7074 of *Lecture Notes in Computer Science*, pages 374–383. Springer, 2011.

[GT88] Andrew V Goldberg and Robert E Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM (JACM)*, 35(4):921–940, 1988.

[Hal87] Philip Hall. On representatives of subsets. *Classic Papers in Combinatorics*, pages 58–62, 1987.

[Har98] David A Harville. Matrix algebra from a statistician's perspective, 1998.

[HK73] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973.

[Hoc08] Dorit S Hochbaum. The pseudoflow algorithm: A new algorithm for the maximum-flow problem. *Operations research*, 56(4):992–1009, 2008.

[HTFF09] Trevor Hastie, Robert Tibshirani, Jerome H Friedman, and Jerome H Friedman. *The elements of statistical learning: data mining, inference, and prediction*, volume 2. Springer, 2009.

[Ind06] Piotr Indyk. Stable distributions, pseudorandom generators, embeddings, and data stream computation. *J. ACM*, 53(3):307–323, 2006.

[JKPS17] Gorav Jindal, Pavel Kolev, Richard Peng, and Saurabh Sawlani. Density independent algorithms for sparsifying k-step random walks. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2017, August 16-18, 2017, Berkeley, CA, USA*, pages 14:1–14:17, 2017.

[JL84] William B Johnson and Joram Lindenstrauss. Extensions of lipschitz mappings into a hilbert space 26. *Contemporary mathematics*, 26, 1984.

[JST11] Hossein Jowhari, Mert Saglam, and Gábor Tardos. Tight bounds for lp samplers, finding duplicates in streams, and related problems. In *Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2011, June 12-16, 2011, Athens, Greece*, pages 49–58, 2011.

[JW21] Rajesh Jayaram and David P. Woodruff. Perfect ℓ_{p} sampling in a data stream. *SIAM J. Comput.*, 50(2):382–439, 2021.

[Kap13] Michael Kapralov. Better bounds for matchings in the streaming model. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 1679–1697, 2013.

[Kap21] Michael Kapralov. Space lower bounds for approximating maximum matching in the edge arrival model. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 1874–1893. SIAM, 2021.

[Kar73] Alexander V Karzanov. On finding maximum flows in networks with special structure and some applications. *Matematicheskie Voprosy Upravleniya Proizvodstvom*, 5:81–94, 1973.

[KKTY21] Michael Kapralov, Robert Krauthgamer, Jakab Tardos, and Yuichi Yoshida. Towards tight bounds for spectral sparsification of hypergraphs. In Samir Khuller and Virginia Vassilevska Williams, editors, *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, pages 598–611. ACM, 2021.

[KLL16] Zohar S. Karnin, Kevin J. Lang, and Edo Liberty. Optimal quantile approximation in streams. In Irit Dinur, editor, *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*, pages 71–78. IEEE Computer Society, 2016.

[KLM+14] Michael Kapralov, Yin Tat Lee, Cameron Musco, Christopher Musco, and Aaron Sidford. Single pass spectral sparsification in dynamic streams. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 561–570, 2014.

[KLM+17] Michael Kapralov, Yin Tat Lee, Cameron Musco, Christopher Musco, and Aaron Sidford. Single pass spectral sparsification in dynamic streams. *SIAM J. Comput.*, 46(1):456–477, 2017.

[KLOS14] 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 *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pages 217–226, 2014.

[KLP+16] Rasmus Kyng, Yin Tat Lee, Richard Peng, Sushant Sachdeva, and Daniel A Spielman. Sparsified cholesky and multigrid solvers for connection laplacians. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 842–850, 2016.

[KM15] Fabian Kuhn and Anisur Rahaman Molla. Distributed sparse cut approximation. In Emmanuelle Anceaume, Christian Cachin, and Maria Gradinariu Potop-Butucaru, editors, *19th International Conference on Principles of Distributed Systems, OPODIS 2015, December 14-17, 2015, Rennes, France*, volume 46 of *LIPIcs*, pages 10:1–10:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015.

[KMM12] Christian Konrad, Frédéric Magniez, and Claire Mathieu. Maximum matching in semi-streaming with few passes. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques - 15th International Workshop, APPROX 2012, and 16th International Workshop, RANDOM 2012, Cambridge, MA, USA, August 15-17, 2012. Proceedings*, volume 7408 of *Lecture Notes in Computer Science*, pages 231–242. Springer, 2012.

[KMM+20] Michael Kapralov, Aida Mousavifar, Cameron Musco, Christopher Musco, Navid Nouri, Aaron Sidford, and Jakab Tardos. Fast and space efficient spectral sparsification in dynamic streams. In *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Al-*

gorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020, pages 1814–1833, 2020.

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

[KN21] Christian Konrad and Kheeran K. Naidu. On two-pass streaming algorithms for maximum bipartite matching. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2021, August 16-18, 2021, University of Washington, Seattle, Washington, USA (Virtual Conference)*, volume 207 of *LIPIcs*, pages 19:1–19:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[Kön16] Dénes König. Über graphen und ihre anwendung auf determinantentheorie und mengenlehre. *Mathematische Annalen*, 77(4):453–465, 1916.

[Kon18] Christian Konrad. A simple augmentation method for matchings with applications to streaming algorithms. In *43rd International Symposium on Mathematical Foundations of Computer Science, MFCS 2018, August 27-31, 2018, Liverpool, UK*, pages 74:1–74:16, 2018.

[Kou14] Ioannis Koutis. Simple parallel and distributed algorithms for spectral graph sparsification. In Guy E. Blelloch and Peter Sanders, editors, *26th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '14, Prague, Czech Republic - June 23 - 25, 2014*, pages 61–66. ACM, 2014.

[KRV06] Rohit Khandekar, Satish Rao, and Umesh V. Vazirani. Graph partitioning using single commodity flows. In Jon M. Kleinberg, editor, *Proceedings of the 38th Annual ACM Symposium on Theory of Computing, Seattle, WA, USA, May 21-23, 2006*, pages 385–390. ACM, 2006.

[KS16] Rasmus Kyng and Sushant Sachdeva. Approximate gaussian elimination for laplacians - fast, sparse, and simple. In Irit Dinur, editor, *IEEE 57th Annual Symposium on*

Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA, pages 573–582. IEEE Computer Society, 2016.

[KSV10] Howard J. Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In Moses Charikar, editor, *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 938–948. SIAM, 2010.

[KT17] Sagar Kale and Sumedh Tirodkar. Maximum matching in two, three, and a few more passes over graph streams. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2017, August 16-18, 2017, Berkeley, CA, USA*, volume 81 of *LIPIcs*, pages 15:1–15:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[KVV04] Ravi Kannan, Santosh S. Vempala, and Adrian Vetta. On clusterings: Good, bad and spectral. *J. ACM*, 51(3):497–515, 2004.

[Lee14] Yin Tat Lee. Probabilistic spectral sparsification in sublinear time. *CoRR*, abs/1401.0085, 2014.

[LGT14] James R. Lee, Shayan Oveis Gharan, and Luca Trevisan. Multiway spectral partitioning and higher-order cheeger inequalities. *J. ACM*, 61(6):37:1–37:30, 2014.

[Li20] Jason Li. Faster parallel algorithm for approximate shortest path. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 308–321, 2020.

[LMP13] Mu Li, Gary L. Miller, and Richard Peng. Iterative row sampling. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*, pages 127–136, 2013.

[LNPS23] Jason Li, Danupon Nanongkai, Debmalya Panigrahi, and Thatchaphol Saranurak.

Near-linear time approximations for cut problems via fair cuts. In Nikhil Bansal and Viswanath Nagarajan, editors, *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023, Florence, Italy, January 22-25, 2023*, pages 240–275. SIAM, 2023.

[LPYZ20] Huan Li, Stacy Patterson, Yuhao Yi, and Zhongzhi Zhang. Maximizing the number of spanning trees in a connected graph. *IEEE Trans. Inf. Theory*, 66(2):1248–1260, 2020.

[LRTV12] Anand Louis, Prasad Raghavendra, Prasad Tetali, and Santosh S. Vempala. Many sparse cuts via higher eigenvalues. In *Proceedings of the 44th Symposium on Theory of Computing Conference, STOC 2012, New York, NY, USA, May 19 - 22, 2012*, pages 1131–1140, 2012.

[LS18] Huan Li and Aaron Schild. Spectral subspace sparsification. In Mikkel Thorup, editor, *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018*, pages 385–396. IEEE Computer Society, 2018.

[LZ18] Huan Li and Zhongzhi Zhang. Kirchhoff index as a measure of edge centrality in weighted networks: Nearly linear time algorithms. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 2377–2396. SIAM, 2018.

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

[Mad16] Aleksander Madry. Computing maximum flow with augmenting electrical flows. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 593–602. IEEE, 2016.

[McG14] Andrew McGregor. Graph stream algorithms: a survey. *SIGMOD Rec.*, 43(1):9–20, 2014.

[MRL99] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G. Lindsay. Random sampling techniques for space efficient online computation of order statistics of large datasets. In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*, pages 251–262. ACM Press, 1999.

[MS21] Bogdan-Adrian Manghiuc and He Sun. Hierarchical clustering: O(1)-approximation for well-clustered graphs. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 9278–9289, 2021.

[MU17] Michael Mitzenmacher and Eli Upfal. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press, 2017.

[Mut05] Shanmugavelayutham Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends® in Theoretical Computer Science*, 1(2):117–236, 2005.

[NS17] Danupon Nanongkai and Thatchaphol Saranurak. Dynamic spanning forest with worst-case update time: adaptive, las vegas, and o($n^{1/2 - \varepsilon}$)-time. In Hamed Hatami, Pierre McKenzie, and Valerie King, editors, *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 1122–1129. ACM, 2017.

[OG21] James B. Orlin and Xiao-Yue Gong. A fast maximum flow algorithm. *Networks*, 77(2):287–321, 2021.

[Pag01] Rasmus Pagh. Low redundancy in static dictionaries with constant query time. *SIAM J. Comput.*, 31(2):353–363, 2001.

[Pen16] Richard Peng. Approximate undirected maximum flows in $O(m\text{polylog}(n))$ time. In

Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016, pages 1862–1867, 2016.

[PF22] Yossi Peretz and Yigal Fischler. A fast parallel max-flow algorithm. *Journal of Parallel and Distributed Computing*, 169:226–241, 2022.

[PR02] Seth Pettie and Vijaya Ramachandran. A randomized time-work optimal parallel algorithm for finding a minimum spanning forest. *SIAM Journal on Computing*, 31(6):1879–1895, 2002.

[PS14] Richard Peng and Daniel A Spielman. An efficient parallel solver for sdd linear systems. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 333–342, 2014.

[Ram90] Vijaya Ramachandran. Parallel algorithms for shared-memory machines. In *Algorithms and Complexity*, pages 869–941. Elsevier, 1990.

[RGH+22] Václav Rozhoň, Christoph Grunau, Bernhard Haeupler, Goran Zuzic, and Jason Li. Undirected $(1+\varepsilon)$-shortest paths via minor-aggregates: near-optimal deterministic parallel and distributed algorithms. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*, pages 478–487, 2022.

[RP16] Aurko Roy and Sebastian Pokutta. Hierarchical clustering via spreading metrics. In Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 2316–2324, 2016.

[RP17] Aurko Roy and Sebastian Pokutta. Hierarchical clustering via spreading metrics. *J. Mach. Learn. Res.*, 18:88:1–88:35, 2017.

[RR03] Rajeev Raman and S. Srinivasa Rao. Succinct dynamic dictionaries and trees. In

Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proceedings, volume 2719 of *Lecture Notes in Computer Science*, pages 357–368. Springer, 2003.

[RST14] Harald Räcke, Chintan Shah, and Hanjo Täubig. Computing cut-based hierarchical decompositions in almost linear time. In Chandra Chekuri, editor, *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pages 227–238. SIAM, 2014.

[RSW18] Aviad Rubinstein, Tselil Schramm, and S. Matthew Weinberg. Computing exact minimum cuts without knowing the graph. In *9th Innovations in Theoretical Computer Science Conference, ITCS 2018, January 11-14, 2018, Cambridge, MA, USA*, pages 39:1–39:16, 2018.

[She09] Jonah Sherman. Breaking the multicommodity flow barrier for o(vlog n)-approximations to sparsest cut. In *50th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2009, October 25-27, 2009, Atlanta, Georgia, USA*, pages 363–372. IEEE Computer Society, 2009.

[She13a] Jonah Sherman. Nearly maximum flows in nearly linear time. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*, pages 263–269, 2013.

[She13b] Jonah Sherman. Nearly maximum flows in nearly linear time. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 263–269. IEEE, 2013.

[She17a] Jonah Sherman. Area-convexity, ℓ_∞ regularization, and undirected multicommodity flow. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 452–460, 2017.

[She17b] Jonah Sherman. Generalized preconditioning and undirected minimum-cost flow. In

Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, pages 772–780. SIAM, 2017.

[SMR08] Hinrich Schütze, Christopher D Manning, and Prabhakar Raghavan. *Introduction to information retrieval*, volume 39. Cambridge University Press Cambridge, 2008.

[SS91] Maria Serna and Paul Spirakis. Tight rnc approximations to max flow. In *STACS 91: 8th Annual Symposium on Theoretical Aspects of Computer Science Hamburg, Germany, February 14–16, 1991 Proceedings 8*, pages 118–126. Springer, 1991.

[SS11a] Daniel A. Spielman and Nikhil Srivastava. Graph sparsification by effective resistances. *SIAM J. Comput.*, 40(6):1913–1926, 2011.

[SS11b] Daniel A Spielman and Nikhil Srivastava. Graph sparsification by effective resistances. *SIAM Journal on Computing*, 40(6):1913–1926, 2011.

[ST04] Daniel A. Spielman and Shang-Hua Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*, pages 81–90, 2004.

[ST14a] Daniel A. Spielman and Shang-Hua Teng. Nearly linear time algorithms for preconditioning and solving symmetric, diagonally dominant linear systems. *SIAM J. Matrix Anal. Appl.*, 35(3):835–885, 2014.

[ST14b] Daniel A. Spielman and Shang-Hua Teng. Nearly linear time algorithms for preconditioning and solving symmetric, diagonally dominant linear systems. *SIAM J. Matrix Anal. Appl.*, 35(3):835–885, 2014.

[SV80] Yossi Shiloach and Uzi Vishkin. An o (log n) parallel connectivity algorithm. Technical report, Computer Science Department, Technion, 1980.

[SV82] Yossi Shiloach and Uzi Vishkin. An o (n2log n) parallel max-flow algorithm. *Journal of Algorithms*, 3(2):128–146, 1982.

[Sze75] Endre Szemerédi. Regular partitions of graphs. Technical report, Stanford Univ Calif Dept of Computer Science, 1975.

[Tro11] Joel Tropp. Freedman's inequality for matrix martingales. *Electronic Communications in Probability*, 16:262–270, 2011.

[Tro12] Joel A. Tropp. User-friendly tail bounds for sums of random matrices. *Found. Comput. Math.*, 12(4):389–434, 2012.

[Waj20] David Wajc. *Matching Theory Under Uncertainty*. PhD thesis, Carnegie Mellon University, 2020.

[Woo14] David P Woodruff. Sketching as a tool for numerical linear algebra. *Foundations and Trends® in Theoretical Computer Science*, 10(1–2):1–157, 2014.

[Yao77] Andrew Chi-Chih Yao. Probabilistic computations: Toward a unified measure of complexity (extended abstract). In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 222–227. IEEE Computer Society, 1977.

[ZGY+22] Goran Zuzic, Gramoz Goranci, Mingquan Ye, Bernhard Haeupler, and Xiaorui Sun. Universally-optimal distributed shortest paths and transshipment via graph-based l1-oblivious routing. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2549–2579. SIAM, 2022.