NORTHWESTERN UNIVERSITY

Fantastic Subgraphs and How to Find Them

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Computer Science

By

Pattara Sukprasert

EVANSTON, ILLINOIS

June 2023

ABSTRACT

Fantastic Subgraphs and How to Find Them

Pattara Sukprasert

Due to their widespread applicability, graphs and networks appear in various contexts. The increasing scale of graphs encountered in the real-world requires the development of efficient algorithms that run reasonably fast and produce close to optimal solutions. The main focus of this thesis is the development of fast graph algorithms for optimizing specific structural properties. Many classical problems can be thought of in this framework - for example, the maximum weight matching problem in graphs can be thought of as finding a maximum weight subgraph where each node has degree at most one. While these problems are known to be solvable in polynomial time, many interesting properties could lead to NP-hard optimization problems. We study four problems that fit in the framework.

First, we study *k-Edge Connected Spanning Subgraph* (kECSS). Our goal here is to come up with a network that tolerates link failures. Given an undirected weighted graph, we want to find a minimum weighted spanning subgraph with a property that this subgraph must remain connected even after removing k edges. This problem has applications in route planning where links such as roads or network cables might become unusable. We can think of it as a generalization to the classical minimum spanning tree problem, and this belongs to a family of problems called Survivable Network Design. It has been known for a long time

how to get a 2-approximation solution in polynomial time. We show how to find a fractional solution in near-linear time and how to round it to an integral solution efficiently.

Second, we study the *Dynamic Graph Spanner* problem. Here, our goal is to develop a data structure, called a *spanner*, that maintains a subgraph approximating distances between pairs of nodes dynamically. A dynamic graph undergoes structural changes such as edge insertions and deletions. More precisely, our goal is to maintain a t-spanner, which is small subgraph such that the distance between every pair of vertices is bounded by t times their actual distance. In the area of dynamic graphs, one important goal is to design algorithms that are robust against an adaptive adversary. Maintaining a dynamic spanner is one of the problems where the gap between an oblivious adversary and an adaptive adversary was not well understood. When *recourse*, which is defined as the number of edge changes, is of concern, we show how to maintain a dynamic spanner with an optimal bound. We further show that we can achieve good runtime and recourse for maintaining a dynamic 3-spanner.

We then focus on a problem related to *Dataset Versioning*. In this problem, our nodes represent versions, and edges represent "deltas" which are differences between versions. This problem captures online collaboration and data repositories. The general goal is to store a selected set of versions under a storage constraint while allowing users to query a certain version efficiently. Here, we are interested in optimizing both weights and distances while ensuring connectivity. Despite research in the graph representation of these problems, directed variants of this problem are poorly understood. We show both hardness results and algorithms for different specific cases.

Lastly, we focus on the *Densest Subgraph* problem. This problem has applications in graph mining and bio-informatics and has recently gained much traction. Given an undirected unweighted graph, the goal is to find a vertex-induced subgraph with maximum density, defined as a ratio between the number of edges and the number of vertices of the graph. We focus on developing a parallel algorithm that runs within a reasonable time on massive-scale graphs for up to billions of nodes.

Acknowledgements

Six years in two graduate schools is a long time. During these years, I have been through good times, bad times, joy, tears, sorrow, stress, transition, and changes. I think I had quite eventful years and I would not be here today without help from many people around me. I am blessed to have such a support system.

First and foremost, I would like to thank Samir Khuller, my advisor. I became Samir's student during my first semester at the University of Maryland in College Park. At that time, I had not graduated with my Master's degree in Thailand yet, so it was like working two jobs simultaneously. Thank you for being patient with me and believing in me. Under his guidance, I learned to be a better student and researcher. I cannot thank Samir enough for the lessons, advice, connections, and support. Samir is well-known in the community and is respected by his peer. Not only that, he is maintaining those networks very well. The mindset I learned will stick with me for the rest of my life. Thank you for setting such a standard and being my role model.

I met two of my committee members, Aravindan Vijayaraghavan and Konstantin Makarychev, when I moved to Northwestern University. Most of our interactions were through our group's events. Thank you for being a good part of giving our group a sense of community. Also, thank you both for agreeing to be on my committee.

I first met Thatchaphol Saranurak as a Master's student in Thailand. He started his Ph.D. around then. Since then, we have been collaborating on different projects, and I got to learn about different problems and trends from him. He also hosted me a few times on other occasions. Thank you for being a senior who took care of me and for being on my committee. I learned a lot from you, especially about dynamic graphs. Our dynamic spanner project became a chapter in this thesis. I always admire your mindset toward research and your effort to ensure your projects are adequately taken care of. It is astonishing to me the sheer amount of results you got throughout the years.

I started working with Julian Shun on the densest subgraph project. The project only started in the Summer of 2022, so this is the most recent part of my thesis. While relatively brief, I immensely enjoyed working on the problem and learned about parallel algorithms. Thank you for funding all the costs of that project and for being on my committee.

I would not have known TCS in the first place without the Thai TCS community. Especially from my Master's advisor, Jittat Fakcharoenphol, whom I worked with at the beginning of my career. I was working full-time as a software engineer and doing research part-time for a year before leaving my job and being a full-time student. Thank you a lot for your support, your inspiration, and for helping me with my Mathematical Maturity , which I lacked before.

I visited a few universities during my study. I thank Parinya Chalermsook and Danupon Nanongkai for hosting me on my research internship at Aalto University and KTH. I thank Bundit Lekhanukit for hosting me at the Shanghai University of Finance and Economics. The research internship was when I worked on the k-Edge Connected Spanning Subgraph problem, which appears in this thesis.

There, at Aalto University, I also got to work on Tuza's conjecture intensively with Sumedha Uniyal, who was a postdoctoral fellow. That project became the first paper published during my doctoral study. Sumedha, I will always treasure the countless hours we spent fixing the bug to make that submission. You will be remembered.

I thank all collaborators whose contributions appear in this thesis. Thank you for your discussion, collaboration, and friendship. For the k-Edge Connected Spanning Subgraph project, I thank Parinya Chalermsook, Chien-Chung Huang, Danupon Nanongkai, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. For the Dynamic Spanner project, I thank Sayan Bhattacharya and Thatchaphol Saranurak. For the Graph Versioning project, I thank Amol Deshpande, Bob Guo, Sofia Li, Koyel Mukherjee, and Samir Khuller. For the Densest Subgraph project, I thank Laxman Dhulipala, Quanquan Liu, and Julian Shun.

Beyond that, I enjoyed hours of discussions with all my colleagues. Thank you for being part of my wonderful experience. For that, I thank Anders Aamand, Saba Ahmadi, Natalie Artzi, Pranjal Awasthi, Justin Y Chen, Keertana Chidambaram, Matthäus Kleindessner, Aounon Kumar, Allen Liu, Jamie Morgenstern, Eytan Ruppin, Alejandro A Schäffer, Fiorella Schischlik, Sanju Sinha, Sandeep Silwal, Sumedha Uniyal, Ali Vakilian, Rahulsimham Vegesna, Sheng Yang, and Fred Zhang.

I made many friends who shared the doctoral journey during my graduate school years. While I cannot list all of you, I would like to thank you all for being different colors in my life. Thank you for the classes, events, talks, meals, trips, fun, and all the time we had. I especially want to mention more than a few names. My two years at Maryland would not be as great without Amin Ghiasi, Rangfu Hu, Shih-Han Hung, Jingling Li, Yuheng Lu, Khoi Pham, and Xuchen You. At Northwestern, I interacted with theory professors, students, and postdocs. Our group is excellent, active, and fun to be with. I am proud to be part of it. I am grateful for the following list, which is not exhaustive. Thank you, Aidao Chen, Sami Davies, Jinshuo Dong, Sanchit Kalhan, Yiduo Ke, Quanquan Liu, Sheng Long, Aravind Reddy, Anant Shah, Liren Shan, Vaidehi Srinivas, Yifan Wu, and Chenhao Zhang. I want to give a special thanks to Quanquan Liu and Vaidehi Srinivas for helping me name my thesis. The memory of Vaidehi dashing to Mudd 3514 to tell me to change my thesis name will stick with me for the rest of my life.

I am fortunate to come across many Thais, and many of them have become my lifelong friends. Thank you all for being a big part of my life outside of school.

Through my partner, Wiriya Thongsomboon, I learned to be less desperate and led a more peaceful life. Thank you so much for your daily support, love, care, and suggestions and for being the best partner I could hope for. Lastly, I want to thank my family, Boonsom Sukprasert, Somsri Sukprasert, and Apichaya Sukprasert, for raising me, supporting me, and continuing to do so throughout different stages of my life. I am truly blessed to have all of you in my life.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

Graphs are mathematical objects that are widely used in various disciplines today. In their simplest form, graphs consist of *vertices*, which are points that represent entities, and *edges*, which represent relationships between these entities. Many real-world problems can be represented using this simple model. For instance, we can model social networks using vertices to represent individuals and edges to represent their friendships. Furthermore, graphs enable us to focus on solving problems at their core by ignoring irrelevant details. An algorithm that finds a short route from Chicago to Boston is essentially the same as an algorithm that finds a short route for sending data from a client to a server. Graph theory dates back to 1736 when Leonhard Euler proved that the Seven Bridges of Königsberg problem has no solution [Eul41]. Since then, several exciting developments have been made, from the Four Color Theorem [Gut80, AH77] to the more recent Graph Neural Networks [ZCH+20, SGT+08, WPC+20].

Due to their widespread applicability, graphs and networks show up in various contexts, from indexing web pages, finding shortest routes, matching with a potential date, and more. In the age of data, the instances we encounter are increasingly larger. Once upon a time, most algorithms were run on inputs entered by hand by a user. However, as networked devices collect data and people interact on online social platforms, the need to process graphs at scale efficiently is ever-increasing. For example, a significant fraction of the world's population is on social media, such as Facebook and Twitter, creating the need to understand how to process graphs with billions of nodes and edges. Moreover, the graphs are dynamic, so the algorithms might need to be rerun frequently.

While fast implementations of classical algorithms have been developed over the last few decades, the last three decades of research in approximation algorithms have focused on developing approximation algorithms for graph problems that run in polynomial time and produce close to optimal solutions. Again, while significant progress was made in the last three decades in this area, we can no longer be satisfied with algorithms that run in "polynomial time" due to the scale of the graphs involved. Unless the algorithm is practical for graphs with billions of edges its applicability could be severely limited.

This thesis is primarily concerned with the development of fast graph algorithms. Specifically, we study problems of finding "fantastic" subgraphs. These are typically subgraphs that have low cost/few edges that have certain desirable properties. Many classical problems can be thought of in this framework - for example, the maximum weight matching problem in graphs can be thought of as finding a maximum weight subgraph such that each node has degree at most one. Similarly, a maximum weight perfect matching problem can be thought of as a matching where each node has degree exactly one. While these problems are known to be solvable in polynomial time, many interesting properties could lead to NP-hard optimization problems.

We study four problems that fit in the framework. The setting where we study each problem is different, which is natural given that the state-of-the-art for each problem is different. We briefly introduce each problem below. More detailed explanations and the status of each problem can be found in the corresponding chapter.

1.1. k-Edge Connected Spanning Subgraphs

The first problem is about finding a low-cost highly-connected subgraph. Given an undirected weighted graph $G = (V, E)$, a weight function w, and a requirement function r. The goal is to find a subgraph $H \subseteq G$ such that there are at least $r(u, v)$ *disjoint paths* between vertices u and v. This class of problems is called *survivable network design problems*

(SNDPs). In this thesis, we focus on *k-edge connected spanning subgraph*, which are specific cases when $r(u,v) = k$ for a fixed integer k. A *k-edge connected graph* is a graph that remains connected even after any set of $k-1$ edges are removed.

Problem 1 (k-Edge Connected Spanning Subgraph (kECSS))**.** *Given a weighted undirected graph $G = (V, E)$, an integer k, and a weight function $w : E \to \mathbb{R}$. The goal is to find a k-edge connected subgraph $H \subseteq G$ that minimizes $w(H)$.*

The kECSS problem is a generalization of the minimum spanning tree problem (MST) as MST corresponds to the case where $r(u,v) = 1$. Real networks are susceptible to failure, e.g., a road is flooded or a network cable is cut. Hence, there is a need to design fault-tolerance networks, which motivates this line of work. While it is known how to solve MST in linear time [KKT95], the problem becomes NP-hard for all $k > 1$. For SNDPs, the best-known algorithm is the iterative rounding algorithm proposed by Jain [Jai01], which is 2-approximation. However, this approach solves linear programs in many iterations, so it is not scalable. For kECSS, the combinatorial algorithm by Khuller and Vishkin [KV94] is 2-approximation and runs in $\tilde{O}(nmk)$.[1]

In Chapter 2, we discuss a fast $(2+\varepsilon)$-approximation algorithm for kECSS. We first give an algorithm that compute a $(1+\varepsilon)$-approximate fractional solution of kECSS that runs in $\tilde{O}_\varepsilon(m)$.[2] We then show how to use this fractional solution to *sparsify* the input graph G. By plugging in the algorithm of Khuller and Vishkin [KV94], we obatin a $(2+\varepsilon)$-approximation algorithm that runs in $\tilde{O}_\varepsilon(m + k^2 n^{1.5})$.

1.2. Dynamic Spanners

The second problem concerns subgraphs that preserve distances approximately; such subgraphs are called *multiplicative spanner*. A subgraph $H \subseteq G$ is a *t-spanner* if $d_G(u,v) \leq$

[1]$\tilde{O}$ hides *polylog*(n) factors.
[2]O_ε hides $f(1/\varepsilon)$ factors.

$d_H(u,v) \leq t \cdot d_G(u,v)$ where d_G is a distance metric with respect to the graph G. This notion was defined by Peleg and Schäffer [PS89] where they showed that deciding if a graph has a t-spanner of less than m edges is NP-hard. Graph spanners have applications in many areas. For example, in communication networks, one can think of a protocol where each node broadcasts messages to all neighbors, which results in a fast message-propagation but edges are unnecessarily flooded with messages. By constructing a spanner, we reduce the bandwidth by paying the price of a higher latency.[3] We focus our attention on maintaining a small t-spanner in dynamic graphs where a single edge insertion or deletion occurs on each timestep.

Problem 2 (Dynamic Spanner)**.** *Given an undirected graph $G = (V, E)$ and an integer t where the graph G is undergoing a sequence of edge updates. The goal is to come up with an efficient algorithm that maintains a t-spanner $H \subseteq G$ after each update.*

Graph Spanner is well-studied in the static graph setting. In particular, we know how to find an *optimal size spanner*.[4] In dynamic graphs, one problem that gains much attention is designing algorithms that work against an adaptive adversary. An adaptive adversary can decide on an update after learning about the latest output. Maintaining spanners is notable here as it is a problem where efficient algorithms against an oblivious adversary are known [BKS12, FG19, BFH19], but there is almost no result that works against an adaptive adversary. Before our result, the best-known algorithm for $t \leq 3$ requires $O(n)$ per update [AFI06].

In Chapter 3, we discuss in detail algorithms for maintaining spanners that work well against an adaptive adversary in detail. There, we first consider the notion of *recourse* where we try to minimize the number of edges changes per update. We manage to show an

[3]See the survey by Ahmedet al. [ABS$^+$20] for more applications.

[4]Assuming the Girth conjecture [Erd86], there is an instance where a $(2k-1)$-spanner requires at least $n^{1+1/k}$ edges. We refer to a spanner of size $O(n^{1+1/k})$ as optimal.

algorithm with small recourse for all t-spanners. We then develop an efficient algorithm that maintains a 3-spanner against an adaptive adversary.

1.3. Dataset Versioning

The third problem is motivated by the need to utilize storage efficiently while keeping all objects accessible in a timely manner. This problem comes up naturally in the context of online collaboration. Imagine a team of scientists working on datasets. Each scientist might manipulate these datasets, transform them, and experiment with them differently, which results in data pipelines. A mistake could happen in an intermediate version of these datasets, and scientists might want to go back to that version to fix it. By explicitly storing every version, it is easy to retrieve a specific version to work as needed. However, doing so is not feasible as it requires much larger storage compared to keeping only some versions. If we keep track of necessary "differences" , i.e., how to construct one version from another, then we can recompute a certain version from versions we have in storage. Deltas here can range from text diffs, SQL-like join operations, to function calls. In this aspect, it is helpful if the time needed to reconstruct each version is not too long. Hence, this motivates *Dataset Versioning*, a bi-criteria optimization problem, where we want to optimize both *storage* and the time needed to reconstruct a specific version, which we call *re-n time*.

Problem 3 (Dataset Versioning)**.** *Given a directed graph $G = (V, E)$. A storage cost function $s : V \cup E \to \mathbb{N}$, a retrieval cost function $r : E \to \mathbb{N}$, the goal is to select a set of vertices $S \subseteq V$ and a set of edges $F \subseteq E$ to minimize*

(1) Total storage cost: $\sum_{v\in S} s(v) + \sum_{e\in F} s(e)$, and

(2) Retrieval cost for each vertex $v \in V \setminus S$, which is the shortest distance from S to v w.r.t. the retrieval cost function r.

Because the problem is bi-criteria, there are natural variants of this problem where we set one criterion as an optimization goal and another as a constraint. The seminal work by Bhattacherjee et al. [BCH+15] defines six different variants and proposes heuristic-based algorithms. In Chapter 4, we discuss algorithms, hardnesses, and experiments concerning these variants. In particular, we show algorithms with a provable guarantee for specific cases where the input graphs are tree-like. We also show reductions to known hard problems, which imply various inapproximability results for these problems.

1.4. Densest Subgraphs

For the last problem, we discuss the problem of finding dense subgraphs, defined as follows.

Problem 4 (Densest Subgraph)**.** *Given an undirected graph $G = (V, E)$, the goal is to find an induced subgraph $H \subseteq G$ that maximizes its density $\rho(H) = |E(H)|/|V(H)|$.*

This is a fundamental problem in graph mining that has been studied for decades. There are numerous applications including social networks visualization [AHDBV05, GJL+13, JXRF09, KRRT99, CHKZ03, RTG14], bioinformatics [FNBB06, SSA+15, DHZ22], and pattern identification [DJD+09, AKS+14, HSB+16].

There exist polynomial-time algorithms for finding the densest subgraph [GGT89, Gol84, Cha00]. However, solving it optimally requires constructing a flow network or solving linear program using solvers. Because of applications in graph mining and bio-informatics, the problem of finding the densest subgraph gained much traction. Unfortunately, the scale of the graphs we are dealing with makes it hard for superlinear time algorithms to be practical. As we aim to get faster algorithms, approximation algorithms are studied. One notable algorithm for finding a 2-approximation solution is the peeling algorithm by Charikar [Cha00]. The algorithm is straightforward. At each step, the algorithm removes ("peels") a vertex with the minimum degree. The algorithm then outputs the graph with the maximum density among all subgraphs the algorithm encountered. The subsequent works by Boob et al. [BGP+20], namely *Greedy++*, improved the quality of the solution by running the peeling algorithm for T iterations, utilizing the idea from the multiplicative weight update framework to update “loads” which is used in addition to the degree to determine peeling order. While these algorithms are already efficient, it lacks parallelism. Also, it is possible to sparsify the graph to speed up algorithms in practice. We propose a pruning rule that can efficiently prune input graphs. We then develop a provably-efficient parallel algorithm for finding the approximate densest subgraph. This will be discussed in Chapter 5.

Putting things in perspective. Among all problems described, the common goal here is to develop algorithms that can obtain a small subgraph while maintaining desired structural property, e.g., a survivable network, a small subgraph approximating the distances, a compressed subgraph, and a dense subgraph. Since the definition of a good subgraph differs from problem to problem, it is natural that the approaches to these problems are different.

Informally, we ask the following question.

Question 1.1. *Given a graph $G = (V, E)$, and a predicate P, find a subgraph $H \subseteq G$ of the smallest size such that H satisfies P.*

1.5. Organization

In Chapter 2, we discuss k-edge connected spanning subgraph. In Chapter 3, we discuss multiplicative spanners. In Chapter 4, we study a problem related to “graph versioning”. In Chapter 5, we discuss the problem of finding dense subgraphs in unweighted graphs. Lastly, we conclude all our works in Chapter 6.

CHAPTER 2

k-Edge Connected Spanning Subgraphs

In the k-Edge-Connected Spanning Subgraph problem (kECSS), we are given an undirected n-node m-edge graph $G = (V, E)$ together with edge costs, and want to find a minimum-cost k-edge connected spanning subgraph.[1] For $k = 1$, this is simply the minimum spanning tree problem, and thus can be solved in $O(m)$ time [KKT95]. For $k \geq 2$, the problem is a classical NP-hard problem whose first approximation algorithm was given almost four decades ago, where Frederickson and Jaja [FJ81] gave a 3-approximation algorithm that runs in $O(n^2)$ time for the case of $k = 2$. The approximation ratio was later improved to 2 by an $\tilde{O}(mnk)$-time algorithm of Khuller and Vishkin [KV94].[2] This approximation factor of 2 has remained the best for more than 30 years, even for a special case of 2ECSS called the weighted tree augmentation problem. When the running time is of the main concern, the fastest known algorithm takes $O(n^2)$ time at the cost of a significantly higher $(2k-1)$-approximation guarantee, due to Gabow, Goemans, and Williamson [GGW98].

This above state-of-the-art leaves a big gap between algorithms achieving the best approximation ratio and the best time complexity. This gap exists even for $k = 2$. In this chapter, we improve the running time of both aforementioned algorithms of [KV94, GGW98] while keeping the approximation ratio arbitrarily close to two. Our main contribution is a near-linear time algorithm that $(1 + \varepsilon)$-approximates the optimal *fractional* solution.

[1]Note that this problem should not be confused with a variant that allows to pick the same edge multiple time, which is sometimes also called kECSS (e.g., [CQ17a]). We follow the convention in [CL07] and call the latter variant minimum-cost k-edge connected spanning sub-multigraph (kECSSM) problem. (See also the work by Pritchard [Pri10].)

[2]$\tilde{O}$ hides polylog(n) factor.

Theorem 2.1. *For any $\varepsilon > 0$, there is a randomized $\widetilde{O}(m/\varepsilon^2)$-time algorithm that outputs a $(1+\varepsilon)$-approximate* fractional *solution for kECSS.*

Following, in the high-level, the arguments of Chekuri and Quanrud [CQ18] (i.e. solving the minimum-weight k disjoint arborescences in the style of [KV94] on the support of the sparsified fractional solution), the above fractional solution can be turned into a fast $(2+\varepsilon)$-approximation algorithm for the integral version of kECSS.

Corollary 2.2. *For any $\varepsilon > 0$, there exist*

- *a randomized $\widetilde{O}(m/\varepsilon^2)$-time algorithm that estimates the* value *of the optimal solution for kECSS to within a factor $(2+\varepsilon)$, and*
- *a randomized $\tilde{O}\left(\frac{m}{\varepsilon^2} + \frac{k^2 n^{1.5}}{\varepsilon^2}\right)$-time algorithm that produces a feasible kECSS solution of cost at most $(2+\varepsilon)$ times the optimal value.*

We remark that the term $\tilde{O}(k^2 n^{1.5})$ is in fact "tight" up to the state-of-the-art algorithm for finding minimum-weight k disjoint arborescences.[3]

Prior to our results, a sub-quadratic time algorithm was not known even for special cases of kECSS, called *k-Edge-Connected Augmentation* (kECA). In this problem, we are given a $(k-1)$-edge-connected subgraph H of a graph G, and we want to minimize the total cost of adding edges in G to H so that H becomes k-edge connected. It is not hard to see that if we can α-approximates kECSS, then we can α-approximates kECA by assigning cost 0 to all edges in H. This problem previously admits a $O(kn^2)$-time 2-approximation algorithm for any even integer k [KT93].[4] The approximation ratio of 2 remains the best even for 2ECA. Our result in Corollary 2.2 improves the previously best time complexity by a $\tilde{\Theta}(\sqrt{n})$ factor.

[3]More formally, if a minimum-weight union of k edge-disjoint arborescences can be found in time $T(k, m, n)$, then our algorithm would run in time $T(k, kn, n)$. The term $O(k^2 n^{1.5})$ came from Gabow's algorithm [Gab95] that runs in time $O(km\sqrt{n}\log(nc_{\max}))$.

[4]In Khuller and Vishkin [KT93], the kECA problem aims at augmenting the connectivity from k to $(k+1)$ (but for us it is from $(k-1)$ to k.)

Perspective. The gap between algorithms with best approximation ratio and best time complexity in fact reflects a general lack of understanding on *fast approximation algorithms*. While polynomial-time algorithms were perceived by many as efficient, it is not a reality in the current era of large data, where it is nearly impossible to take $O(n^3)$ time to process a graph with millions or billions of nodes. Research along this line includes algorithms for sparsest cut [KRV09, KKOV07, She09, Mad10a], multi-commodity flow [GK07, Fle00, Mad10b], and travelling salesman problem [CQ17a, CQ18]. Some of these algorithms have led to exciting applications such as fast algorithms for max-flow [She13], dynamic connectivity [NSW17, CGL+19, SW19, Wul17, NS17a], vertex connectivity [LNP+21] and maximum matching [vdBLN+20].

The kECSS problem belongs to the class of survivable network design problems (SNDPs), where the goal is to find a subgraph ensuring that every pair of nodes (u, v) are $\kappa(u, v)$-edge-connected for a given function κ. (kECSS is the *uniform* version of SNDP where $\kappa(u, v) = k$ for every pair (u, v).) These problems typically focus on building a network that is resilient against device failures (e.g. links or nodes), and are arguably among the most fundamental problems in combinatorial optimization. Research in this area has generated a large number of beautiful algorithmic techniques during the 1990s, culminating in the result of Jain [Jai01] which gives a 2-approximation algorithm for the whole class of SNDPs. Thus, achieving a *fast* 2-approximation algorithm for SNDPs is a very natural goal.

Towards this goal and towards developing fast approximation algorithms in general, there are two common difficulties:

(1) Many approximation algorithms inherently rely on solving a *linear program (LP)* to find a fractional solution, before performing rounding steps. However, the state-of-the-art general-purpose linear program solvers are still quite slow, especially for kECSS and SNDP where the corresponding LPs are *implicit*.

In the context of SNDP, the state-of-the-art (approximate) LP solvers still require at least quadratic time: Fleischer [Fle04] designs an $\tilde{O}(mnk)$ for solving kECSS LP, and more generally for SNDP and its generalization [Fle04, FKPS16] with at least $\Theta(m \min\{n, k_{\max}\})$ iterations of minimum cost flow's computation are the best known running time where $k_{\max}$ is the maximum connectivity requirements.

(2) Most existing techniques that round fractional solutions to integral ones are not "friendly" for the design of fast algorithms. For instance, Jain's celebrated iterative rounding [Jai01] requires solving the LP $\Omega(m)$ times. Moreover, most LP-based network design algorithms are fine-tuned to optimize approximation factors, while designing near-linear time LP rounding algorithms requires limiting ourselves to a relatively small set of tools, about which we currently have very limited understanding.

This work completely resolves the first challenge for kECSS and manages to identify a fundamental bottleneck of the second challenge.

Challenges for LP Solvers. Our main challenge is handling the so-called *box constraints* in the LPs. To be concrete, below is the LP relaxation of kECSS on graph $G = (V, E)$.

$$\min\{\sum_{e\in E} c_e x_e : \sum_{e\in\delta_G(S)} x_e \geq k \ (\forall S \subseteq V), x \in [0,1]^E\} \tag{2.1}$$

where $\delta_G(S)$ is the set of edges between nodes in S and $V \setminus S$. The box constraints refer to the constraints $x \in [0,1]^E$. Without these constraints, we can select the same edge multiple times in the solution; this problem is called kECSSM in [CL07] (see Footnote 1). Removing the box constraints often make the problem significantly easier as the linear program, without box constraints, is often pure covering or pure packing problem. There is a line of work focusing on developing fast algorithms for solving these two classes [PST91, BI04, Jan01, CHPQ20, KY99, CQ17b]. For example, the min-cost st-flow problem without the box constraints become computing the shortest st-path, which admits a much faster algorithm.

For kECSS, it can be shown that solving (2.1) without the box constraints can be reduced to solving (2.1) *with* $k = 1$ and multiplying all x_e with k. In other words, without the box constraints, fractional kECSS is equivalent to fractional 1ECSS.This *fractional* 1ECSS can be $(1+\varepsilon)$-approximated in near-linear time by plugging in the dynamic minimum cut data structure of Chekuri and Quanrud [CQ17a] to the multiplicative weight update framework (MWU).

However, with the presence of box constraints, to use the MWU framework we would need a dynamic data structure for a much more complicated cut problem, that we call, the *minimum normalized free cut* problem (roughly, this is a certain normalization of the minimum cut problem where the costs of up to k heaviest edges in the cut are ignored.) For our problem, the best algorithm in the static setting we are aware of (prior to this work) is to use Zenklusen's $\tilde{O}(mn^4)$-time algorithm [Zen14] for the *connectivity interdiction* problem.[5] This results in an $\tilde{O}(kmn^4)$-time static algorithm. Speeding up and dynamizing this algorithm seems very challenging. Our main technical contribution is an efficient dynamic data structure (in the MWU framework) for the $(1+\varepsilon)$-approximate minimum normalized free cut problem. We explain the high-level overview of our techniques in Section 2.1.

Further Related Works. The kECSS and its special cases have been studied extensively. For all $k \geq 2$, the kECSS problem is known to be APX-hard [Fer98] even on bounded-degree graphs [CKK02] and when the edge costs are 0 or 1 [Pri10]. Although a factor 2 approximation for kECSS has not been improved for almost 3 decades, various special cases of kECSS admit better approximation ratios (see for instance [GKZ18, FGKS18, Adj18]). For instance, the unit-cost kECSS ($c_e = 1$ for all $e \in E$) behaves very differently, admitting a $(1 + O(1/k))$ approximation algorithm [GGTW09, LGS12]. For the 2ECA problem, one

[5]In the connectivity interdiction problem, we are given $G = (V, E)$ and $k \in \mathbb{N}$, our goal is to compute $F \subseteq E$ to delete from G in order to minimize the minimum cut in the resulting graph.

can get a better than 2 approximation when the edge costs are bounded [Adj18, FGKS18]. Otherwise, for general edge costs, the factor of 2 has remained the best known approximation ratio even for the 2ECA problem.

The kECSS problem in special graph classes has also received a lot of attention. In Euclidean setting, a series of papers by Czumaj and Lingas led to a near-linear time approximation schemes for constant k [CL00, CL99]. The problem is solvable in near-linear time when k and treewidth are constant [BG07, CDE$^+$18]. In planar graphs, 2ECSS, 2ECSSM and 3ECSSM admit a PTAS [CGSZ04, BDT14].

Organization. We provide a high-level overview of our proofs in Section 2.1. In Section 5.1, we explain the background on Multiplicative Weight Updates (MWU) for completeness (although this paper is written in a way that one can treat MWU as a black box). In Section 2.3, we prove our main technical component. In Section 2.4, we present our LP solver. In Section 2.5, we show how to round the fractional solution obtained from the LP solver. Due to space limitations, many proofs are deferred to Appendix.

2.1. Overview of Techniques

In this section, we give a high-level overview of our techniques in connection to the known results. Our work follows the standard Multiplicative Weight Update (MWU) framework together with the Knapsack Covering (KC) inequalities (see Section 5.1 for more background). Roughly, in this framework, in order to obtain a near-linear time LP solver for kECSS, it suffices to provide a fast dynamic algorithm for a certain optimization problem (often called the oracle problem in the MWU literature):

Definition 2.3 (Minimum Normalized Free Cuts)**.** *We are given a graph $G = (V, E)$, weight function $\boldsymbol{w} : E \to \mathbb{R}_{\geq 0}$, integer k, and our goal is to compute a cut $S \subseteq V$ together with* *edges $F \subseteq \delta_G(S) : |F| \leq k-1$ that minimizes the following objective:*[6]

$$\min_{S \subsetneq V, F \subseteq \delta_G(S) : |F| \leq (k-1)} \frac{\boldsymbol{w}(\delta_G(S) \setminus F)}{k - |F|},$$

where $\delta_G(S)$ denotes the set of edges that has exactly one end point in S. We call the minimizer (S, F) the minimum normalized free cut.

This is similar to the minimum cut problem, except that we are allowed to "remove" up to $(k-1)$ edges (called *free edges*) from each candidate cut $S \subseteq V$, and the cost would be "normalized" by a factor of $(k - |F|)$.[7] Notice that there are (apparently) two sources of complexity for this problem. First, we need to find the cut S and second, given S, to compute the optimal set $F \subseteq \delta_G(S)$ of free edges. To the best of knowledge, the fastest known algorithm for this problem takes $\tilde{O}(mn^4)$ time by reducing it to the connectivity interdiction problem [Zen14], while we require near-linear time. This is our first technical challenge.

Our second challenge is as follows. To actually speed up the whole MWU framework, in addition to solving the oracle problem statically efficiently, we further need to implement a dynamic version of the oracle with polylog(n) update time. In our case, the goal is to maintain a dynamic data structure on graph $G = (V, E)$, weight function $\mathbf{w}$, cost function c, that supports the following operation:

Definition 2.4. *The* PunishMin *operation computes a $(1 + O(\varepsilon))$-approximate normalized free cut and multiply the weight of each edge $e \in \delta_G(S) \setminus F$ by a factor of at most e^{ε}.*[8]

We remark that invoking the PunishMin operation does not return the cut (S, F), and the only change is the weight function $\mathbf{w}$ being maintained by the data structure.

[6]For any function f, for any subset S of its domain, we define $f(S) = \sum_{s \in S} f(s)$.

[7]This is in fact a special case of a similar objective considered by Feldmann, Könemann, Pashkovich and Sanità [FKPS16], who considered applying the MWU framework for the generalized SNDP.

[8]The actual weight $w(e)$ is updated for all $e \in \delta_G(S) \setminus F$: $w(e) \leftarrow w(e) \cdot \exp(\frac{\varepsilon c_{\min}}{c_e})$ where $c_{\min}$ is the minimum edge capacity in $\delta_G(S) \setminus F$.

Proposition 2.5 (Informal). *Assume that we are given a dynamic algorithm that supports* PunishMin *with amortized* polylog(n) *cost per operations, then the kECSS LP can be solved in time $\tilde{O}(m)$.*

Let us call such a dynamic algorithm a fast **dynamic punisher**. The fact that a fast dynamic punisher implies a fast LP solver is an almost direct consequence of MWU [GK07].

Therefore, we focus on designing a fast dynamic algorithm for solving (and punishing) the minimum normalized free cut problem. Our key idea is an efficient and dynamic implementation of the **weight truncation** idea.

Weight truncation: Let $G = (V, E)$ and $\rho \in \mathbb{R}_{\geq 0}$ be a threshold. For any weight function $\mathbf{w}$ of G, denote by $\mathbf{w}_\rho$ the truncated weight defined by $\mathbf{w}_\rho(e) = \min\{\mathbf{w}(e), \rho\}$ for each $e \in E$. Call an edge e with $\mathbf{w}(e) \geq \rho$ a ρ-heavy edge.

Our main contribution is to show that, when allowing $(1 + \varepsilon)$-approximation, we can use the weight truncation to reduce the minimum normalized free cut to minimum cut with $O(\text{polylog}(n))$ extra factors in the running time. Moreover, this reduction can be implemented efficiently in the dynamic setting. We present the ideas in two steps, addressing our two technical challenges mentioned above respectively. First, we show how to solve the static version of minimum normalized free cut in near-linear time. Second, we sketch the key ideas to implement them efficiently in the dynamic setting, which can be used in the MWU framework.

We remark that weight truncation technique has been used in different context. For instance, Zenklusen [Zen14] used it for reducing the connectivity interdiction problem to $O(|E|)$ instances of the minimum budgeted cut problem.

2.1.1. Step 1: Static Algorithm

We show that the minimum normalized free cut problem can be solved efficiently in the static setting. For convenience, we often use the term cut to refer to a set of edges instead of a set of vertices.

Define the objective function of our problem as, for any cut C,

$$\mathsf{val}_{\mathbf{w}}(C) = \min_{F \subseteq C : |F| \leq k-1} \frac{\mathbf{w}(C \setminus F)}{k - |F|}.$$

For any weight function $\mathbf{w}$, denote by $\mathsf{OPT}_{\mathbf{w}} = \min_C \mathsf{val}_{\mathbf{w}}(C)$. In this paper, the graph G is always fixed, while $\mathbf{w}$ is updated dynamically by the algorithm (so we omit the dependence on G from the notation val and OPT). When $\mathbf{w}$ is clear from context, we sometimes omit the subscript $\mathbf{w}$.

We show that the truncation technique can be used to establish a connection between our problem and minimum cut.

Lemma 2.6. *We are given a graph $G = (V, E)$, weight function $\boldsymbol{w}$, integer k, and $\varepsilon > 0$. For any threshold $\rho \in (\mathsf{OPT}_{\boldsymbol{w}}, (1+\varepsilon)\mathsf{OPT}_{\boldsymbol{w}}]$,*

- *any optimal normalized free cut in $(G, \boldsymbol{w})$ is a $(1+\varepsilon)$-approximate minimum cut in $(G, \boldsymbol{w}_\rho)$, and*
- *any minimum cut C^* in $(G, \boldsymbol{w}_\rho)$ is a $(1+\varepsilon)$-approximation for the minimum normalized free cut.*

Proof. First, consider any cut C with $\mathsf{val}(C) = \mathsf{OPT}$. Let $F \subseteq C$ be an optimal set of free edges for C, so we have $\mathbf{w}_\rho(C \setminus F) \leq \mathbf{w}(C \setminus F) = (k - |F|)\mathsf{OPT}$. Moreover, $\mathbf{w}_\rho(F) \leq |F|\rho$. This implies that

$$\mathbf{w}_\rho(C) = \mathbf{w}_\rho(C \setminus F) + \mathbf{w}_\rho(F) < k\rho \tag{2.2}$$

Next, we prove that any cut in $(G, \mathbf{w}_\rho)$ is of value at least $k\mathsf{OPT}$ (so the cut C is a $(1+\varepsilon)$ approximate minimum cut). Assume for contradiction that there is a cut C' such that $\mathbf{w}_\rho(C') < k\mathsf{OPT}$. Let $F' \subseteq C'$ be the set of ρ-heavy edges. Observe that $|F'| \leq k-1$ since otherwise the total weight $\mathbf{w}_\rho(C')$ would have already exceeded $k\mathsf{OPT}$. This implies that $\mathbf{w}(C' \setminus F') = \mathbf{w}_\rho(C' \setminus F') < (k - |F'|)\mathsf{OPT}$ and that

$$\mathsf{val}(C') \leq \frac{\mathbf{w}(C' \setminus F')}{(k - |F'|)} < \mathsf{OPT}$$

which is a contradiction. Altogether, we have proved the first part of the lemma.

To prove the second part of the lemma, consider a minimum cut C^* in $(G, \mathbf{w}_\rho)$, we have that $\mathbf{w}_\rho(C^*) < \mathbf{w}_\rho(C) < k\rho$ (from Equation (2.2)). Again, the set of heavy edges $F^* \subseteq C^*$ can contain at most $k-1$ edges, so we must have $\mathbf{w}(C^* \setminus F^*) < (k - |F^*|)\rho \leq (k - |F^*|)(1+\varepsilon)\mathsf{OPT}$, implying that $\mathsf{val}(C^*) < (1+\varepsilon)\mathsf{OPT}$. □

We remark that this reduction from the minimum normalized free cut problem to the minimum cut problem does not give an exact correspondence, in the sense that a minimum cut in $(G, \mathbf{w}_\rho)$ cannot be turned into a minimum normalized free cut in $(G, \mathbf{w})$. In other words, the approximation factor of $(1+\varepsilon)$ is unavoidable.

Theorem 2.7. *Given a graph $G = (V, E)$ with weight function $\boldsymbol{w}$ and integer k, the minimum normalized free cut problem can be $(1+\varepsilon)$ approximated by using $O(\frac{1}{\varepsilon} \cdot \log n)$ calls to the exact minimum cut algorithm.*

Proof. We assume that the minimum normalized free cut of G is upper bounded by some value M which is polynomial in $n = |V(G)|$ (we show how to remove this assumption in Appendix A.2). For each i such that $(1+\varepsilon)^i \leq M$, we compute the minimum cut C_i in $(G, \mathbf{w}_{\rho_i})$ where $\rho_i = (1+\varepsilon)^i$ and return one with minimum value $\mathsf{val}(C_i)$. Notice that there must be some i^* such that $\rho_{i^*} \in (\mathsf{OPT}_\mathbf{w}, (1+\varepsilon)\mathsf{OPT}_\mathbf{w}]$ and by the lemma, we must have that C_{i^*} is a $(1+\varepsilon)$-approximate solution for the normalized free cut problem. □

By using any near-linear time minimum cut algorithm e.g., [Kar00], the collorary follows.

Corollary 2.8. *There exists a $(1+\varepsilon)$ approximation algorithm for the minimum normalized free cut problem that runs in time $\tilde{O}(|E|/\varepsilon)$.*

2.1.2. Step 2: Dynamic Algorithm

The next idea we use is from Chekuri and Quanrud [CQ17a]. One of the key concepts there is that it is sufficient to solve a "range punishing" problem in near-linear time; for completeness we prove this sufficiency in Appendix. In particular, the following proposition is a consequence of their work:

Definition 2.9. *A* ***range punisher***[9] *is an algorithm that, on any input graph G, initial weight function $\boldsymbol{w} = \boldsymbol{w}^{\text{init}}$, real numbers ε, and $\lambda \leq \mathsf{OPT}_{\boldsymbol{w}^{\text{init}}}$, iteratively applies* PunishMin *on $(G, \boldsymbol{w})$ until the optimal becomes at least $\mathsf{OPT}_{\boldsymbol{w}} \geq (1+\varepsilon)\lambda$.*

The following proposition connects a fast range punisher to a fast LP solver.

Proposition 2.10. *If there exists a range punisher running in time*

$$\widetilde{O}\left(|E| + K + \sum_{e\in E} \log(\frac{\boldsymbol{w}(e)}{\boldsymbol{w}^{\text{init}}(e)})\right)$$

where K is the number of cuts punished, then, there exists a fast dynamic punisher, and consequently the kECSS LP can be solved in near-linear time.

This proposition applies generally in the MWU framework independent of problems. That is, for our purpose of solving kECSS LP, we need a fast range punisher for the minimum normalized free cut problem. For Chekuri and Quanrud [CQ17a], they need such an algorithm for the minimum cut problem (therefore a fast LP solver for the Held-Karp bound).

[9] Our range punisher corresponds to an algorithm of Chekuri and Quanrud [CQ18] in one epoch.

Theorem 2.11 ([CQ17a], informal)**.** *There exists a fast range punisher for the minimum cut problem.*

Our key technical tool in this paper is a more robust reduction from the range punishing of normalized free cuts to the one for minimum cuts. This reduction works for all edge weights and is suitable for the dynamic setting. That is, it is a strengthened version of Lemma 2.6 and is summarized below (see its proof in Section 2.3).

Theorem 2.12 (Range Mapping Theorem)**.** *Let $(G = (V, E), \boldsymbol{w})$ be a weighted graph. Let $\lambda > 0$ and $\rho = (1 + \gamma)\lambda$.*

(1) If the value of optimal normalized free cut is in $[\lambda, (1 + \gamma)\lambda)$, then the value of minimum cut in $(G, \boldsymbol{w}_\rho)$ lies in $[k\rho/(1 + \gamma), k\rho)$.

(2) For any cut C where $\boldsymbol{w}_\rho(C) < k\rho$, then $\frac{\boldsymbol{w}(C \setminus F)}{k-|F|} < (1 + \gamma)\lambda$, where F contains all ρ-heavy edges in C. In particular, $\mathsf{val}(C) < (1 + \gamma)\lambda$.

Given the above reduction, we can implement range punisher fast. We present its full proof in Section 2.4 and sketch the argument below.

Theorem 2.13. *There exists a fast range punisher for the minimum normalized free cut problem.*

Proof. (sketch) We are given λ and weighted graph $(G, \mathbf{w}) : \mathbf{w} = \mathbf{w}^{\text{init}}$ such that $\mathsf{OPT}_{\mathbf{w}^{\text{init}}} \geq \lambda$. Our goal is to punish the normalized free cuts until the optimal value in $(G, \mathbf{w})$ becomes at least $(1 + \varepsilon)\lambda$. We first invoke Theorem 2.7 to get a $(1 + \varepsilon)$-approximate cut, and if the solution is already greater than $(1 + \varepsilon)^2\lambda$, we are immediately done (this means $\mathsf{OPT} > (1 + \varepsilon)\lambda$).

Now, we know that $\mathsf{OPT} \leq (1 + \varepsilon)^2\lambda \leq (1 + 3\varepsilon)\lambda$. We invoke Lemma 2.12(1) with $\gamma = 3\varepsilon$. The minimum cut in $(G, \mathbf{w}_\rho)$ has size in the range $[k\rho/(1+3\varepsilon), k\rho)$. We invoke (one iteration of) Theorem 2.11 with $\lambda' = k\rho(1 + 3/\varepsilon)$ to obtain a cut C whose size is less than

$k\rho$ and therefore, by Lemma 2.12(1), $\mathsf{val}(C) < (1+3\varepsilon)\lambda$. This is a cut that our algorithm can punish (we ignore the detail of how we actually punish it – we would need to do that implicitly since the cut itself may contain up to m edges). We repeat this process until all cuts whose values are relevant have been punished, that is, we continue this process until the returned cut C has size at least $k\rho$.

The running time of this algorithm is

$$\widetilde{O}\left(|E| + K + \sum_{e\in E}\log(\frac{\mathbf{w}_\rho(e)}{\mathbf{w}_\rho^{\text{init}}(e)})\right) \leq \widetilde{O}\left(|E| + K + \sum_{e\in E}\log(\frac{\mathbf{w}(e)}{\mathbf{w}^{\text{init}}(e)})\right)$$

Notice that we rely crucially on the property of our reduction using truncated weights. □

We remark that in the actual proof of Theorem 2.13, there are quite a few technical complications (e.g., how to find optimal free edges for a returned cut C?), and we cannot invoke Theorem 2.11 in a blackbox manner. We refer to Section 2.4 for the details.

2.1.3. LP Rounding for kECSS

Most known techniques for kECSS (e.g. [GGW98, LGS12]) rely on iterative LP rounding, which is computationally expensive. We achieve fast running time by making use of the 2-approximation algorithm of Khuller and Vishkin [KT93].

Roughly speaking, this algorithm creates a directed graph H from the original graph G and then compute on H the minimum-weight k disjoint arboresences. The latter can be found by Gabow's algorithms, in either $\tilde{O}(|E||V|k)$ or $\tilde{O}(k|E|\sqrt{|V|}\log c_{\max})$ time.

To use their algorithm, we will construct H based on the support of the fractional solution x computed by the LP solver. By the integrality of the arborescence polytope [Sch03], an integral solution is as good as the fractional solution. However, the support of x can be potentially large, which causes Gabow's algorithm to take longer time. Here our idea is a sparsification of the support, by extending the celebrated sparsification theorem of Benzcur and Karger [BK15] to handle our problem, i.e., we prove the following (see Section 2.5 for the proofs):

Theorem 2.14. *Let G be a graph and c_G its capacities. There exists a capacitated graph (H, c_H) on the same set of vertices that can be computed in $\tilde{O}(m)$ such that (i) $|E(H)| = \tilde{O}(nk)$, and (ii) for every cut S and $F \subseteq S : |F| \leq (k-1)$, we have $c_G(S \setminus F) = (1 \pm \varepsilon)\, c_H(S \setminus F)$.*

Benzcur and Karger's theorem corresponds to this theorem when $k = 1$. We believe that this theorem might have further applications, e.g., for providing a fast algorithm for the connectivity interdiction problem. Our result implies the following (see Section 2.5 for the proof):

Theorem 2.15. *Assume that there exists an algorithm that finds a minimum-weight k-arborescences in an m-edge n-node graph in time $T_k(m, n)$. Then there exists a $(2 + \varepsilon)$ approximation algorithm for kECSS running in time $\tilde{O}(m/\varepsilon^2 + T_k(kn/\varepsilon^2, n))$*

Applying Theorem 2.15 with the Gabow's algorithm (see Theorem 2.34 in Section 2.5), we obtain Corollary 2.2.

2.2. Preliminaries

In this section, we review the multiplicative-weight update (MWU) framework for solving a (covering) LP relaxation of the form $\min\{c \cdot x : Ax \geq 1, x \geq 0\}$, where A is an m-by-n matrix with non-negative entries and $c \in \mathbb{R}^n_{\geq 0}$. Our presentation abstracts away the detail of MWU, so readers should feel free to skip this section.

Let $A_1, \ldots, A_m$ be the rows of matrix A. Here is a concrete example:

- **Held-Karp Bound:** The Held-Karp bound on input (G, c) aims at solving the LP:[10]

$$\min\{\sum_{e \in E(G)} c_e x_e : \sum_{e \in S} x_e \geq 2 \text{ for any cut } S \subseteq E\}$$

[10]We refer the readers to [CQ17a] for more discussion about this LP and Held-Karp bound.

Matrix $A = A_G$ is a cut-edge incidence matrix of graph G where each row A_j corresponds to a cut $F_j \subseteq E(G)$, so there are exponentially many rows. Each column corresponds to an edge $e \in E(G)$. There are exactly $|E(G)|$ columns. The matrix is implicitly given as an input graph G.

We explain the MWU framework in terms of matrices. Some readers may find it more illustrative to work with concrete problems in mind.

MWU Framework for Covering LPs: In the MWU framework for solving covering linear programs, we are given as input an m-by-n matrix A and cost vectors c associated with the columns.[11] Let $\varepsilon > 0$ be a parameter; that is, we aim at computing a solution x that is $(1+\varepsilon)$ approximation of the optimal LP solution. Denote by $\textsc{MinRow}(A, w)$ the value $\min_{j\in[m]} A_j w$. We start with an initial weight vector $\mathbf{w}_i^{(0)} = 1/c_i$ for $i \in [n]$. On each day $t = 1, \ldots, T$, we compute an approximately "cheapest" row j^* such that $A_{j^*}\mathbf{w}^{(t-1)} \leq (1+\varepsilon)\textsc{MinRow}(A, \mathbf{w}^{(t-1)})$, and update the weight $\mathbf{w}_i^{(t)} \leftarrow \mathbf{w}_i^{(t-1)}\exp\left(\frac{\varepsilon A_{j^*,i} c_{\min}}{c_i}\right)$ where $c_{\min} = \min_{i\in[n]} \frac{c_i}{A_{j^*,i}}$.[12] After $T = O(n \log n/\varepsilon^2)$ many days, the solution can be found by taking the best scaled vectors; in particular, observe that, for any day t, the scaled vector $\bar{\mathbf{w}}^{(t)} = \mathbf{w}^{(t)}/\left(\min_{j\in[m]} A_j \mathbf{w}^{(t)}\right)$ is always feasible for the LP. The algorithm returns $\bar{\mathbf{w}}^{(t)}$ which has minimum cost. The following theorem shows that at least one such solution is near-optimal.

Theorem 2.16. *For $T = O(\frac{n \log n}{\varepsilon^2})$, one of the solutions $\bar{\mathbf{w}}^{(t)}$ for $t \in [T]$ is a $(1 + O(\varepsilon))$ approximation of the optimal solution* $\min\{c \cdot x : Ax \geq 1, x \geq 0\}$.

Since we use slightly different language than the existing proofs in the literature, we provide a proof in the appendix.

KC Inequalities: Our LP is hard to work with mainly because of the mixed packing/-covering constraints $x \in [0,1]^n$. There is a relatively standard way to get rid of the mixed

[11]There are several ways to explain such a framework. Chekuri and Quanrud [CQ17a] follow the continuous setting of Young [You14]. We instead follow the combinatorial interpretation of Garg and Könemann [GK07].
[12]In the MWU literature, this is often referred to as an oracle problem.

packing/covering constraints by adding Knapsack covering (KC) inequalities into the LP. In particular, for each row (or constraint) $j \in [m]$, we introduce constraints:

$$(\forall F \subseteq \mathsf{supp}(A_j), |F| \leq (k-1)) : \sum_{i \in [n] \setminus F} A_{j,i} x_i \geq k - |F|, \text{ or } \sum_{i \in [n] \setminus F} \frac{A_{j,i}}{(k - |F|)} x_i \geq 1$$

Let A^{kc} be the new matrix after adding KC inequalities, that is, imagine the row indices of A^{kc} as (j, F) where $j \in [m]$ and $F \subseteq \mathsf{supp}(A_j)$; we define $A^{\mathsf{kc}}_{(j,F),i} = A_{j,i}/(k - |F|)$. The actual number of rows in A^{kc} can be as high as $m \cdot n^{O(k)}$, but our algorithm will not be working with this matrix explicitly.

The following lemma shows that we can now remove the packing constraints.

Lemma 2.17. *Any solution to* $\{x \in \mathbb{R}^n : A^{kc}x \geq 1, x \geq 0\}$ *is feasible for* $\{x \in \mathbb{R}^n : Ax \geq k, x \in [0,1]\}$*. Conversely, for any point* z *in the latter polytope, there exists a point* z' *in the former such that* $z' \leq z$*.*

Proof. Let x be a feasible solution $A^{kc}x \geq 1$. Consider $x'_i = \min(x_i, 1)$ for each $i \in [n]$. We claim that x' satisfies $Ax' \geq \kappa$. Consider the constraint $A_j x' \geq \kappa_j$. Let $F = \{i \in \mathsf{supp}(A_j) : x_i > 1\}$. If $|F| \geq \kappa_j$, it would imply that $A_j x' \geq \kappa_j$ and we are done. Otherwise, we have $|F| \leq \kappa_j - 1$, and the KC constraints guarantee that

$$\sum_{i \in \mathsf{supp}(A_j)} x'_i = \sum_{i \in \mathsf{supp}(A_j) \setminus F} x_i + |F| \geq \kappa_j$$

Conversely, let x be a feasible solution $Ax \geq \kappa, x \in [0,1]^n$. Consider any KC constraint: For any $j \in [m]$ and $F \subseteq \mathsf{supp}(A_j), |F| \leq \kappa_j - 1$

$$\sum_{i \in \mathsf{supp}(A_j) \setminus F} x_i = \sum_{i \in \mathsf{supp}(A_j)} x_i - \sum_{i \in F} x_i \geq \kappa_j - |F|$$

This implies that x itself is feasible for $A^{kc}x \geq 1$. □

Corollary 2.18. *For any cost vector $c \in \mathbb{R}^n_{\geq 0}$,*

$$\min\{c^T x : A^{kc} x \geq 1, x \geq 0\} = \min\{c^T x : Ax \geq k, x \in [0,1]\}$$

2.3. Range Mapping Theorem

The goal of this section is to prove Theorem 2.12, a cornerstone of our work. We emphasize that it works for *any* weight function $\mathbf{w}$. First, we introduce more notations for convenience. For any cut $C \in \mathcal{C}$, and any subset of edges $F \subseteq E$, we define $\mathsf{val}_{\mathbf{w}}(C, F) = \frac{\mathbf{w}(C \setminus F)}{k - |F|}$ if $F \subseteq C$ and $|F| < k$; otherwise, $\mathsf{val}_{\mathbf{w}}(C, F) = \infty$. Also, denote $\mathsf{val}_{\mathbf{w}}(C) = \min_{F \subseteq E} \mathsf{val}_{\mathbf{w}}(C, F)$. By definition, we have $\mathsf{val}_{\mathbf{w}}(C) = \min_{i \leq k-1} \mathsf{val}_{\mathbf{w}}(C, F_i)$ where F_i is the set of heaviest i edges in C with respect to weight function $\mathbf{w}$. We let $\mathsf{mincut}_{\mathbf{w}_\rho}$ be the value of a minimum cut with respect with weight $\mathbf{w}_\rho$. When it is clear from context, we sometimes omit the subscript $\mathbf{w}$. For any positive number ρ, let $H_{\mathbf{w},\rho} = \{e \in E \colon \mathbf{w}(e) \geq \rho\}$ be the set of ρ-heavy edges.[13] Define the weight truncation $\mathbf{w}_\rho(e) = \min\{\mathbf{w}(e), \rho\}$.

Theorem 2.19 (Restatement of Theorem 2.12)**.** *We are given a weighted graph $(G, \boldsymbol{w})$, $\lambda > 0$ be a parameter and $\rho = (1+\gamma)\lambda$. Then we have the following:*

(1) If $\mathsf{OPT}_{\boldsymbol{w}} \in [\lambda, (1+\gamma)\lambda)$, then $\mathsf{mincut}_{\boldsymbol{w}_\rho} \in [k\rho/(1+\gamma), k\rho)$, and

(2) if a cut C satisfies $\boldsymbol{w}_\rho(C) < k\rho$, then $\mathsf{val}_{\boldsymbol{w}}(C, H_{\boldsymbol{w},\rho} \cap C) < (1+\gamma)\lambda$.

Notice that the above theorem not only gives a mapping between solutions of the two problems but also that the heavy edges can be used as a set of free edges. We say that a cut C is *interesting* if it contains at most $k-1$ heavy edges, i.e., $|H_{\mathbf{w},\rho} \cap C| < k$.

Proposition 2.20. *If cut $C \subseteq E$ is not interesting (i.e., $|H_{\boldsymbol{w},\rho} \cap C| \geq k$), then $\mathsf{val}_{\boldsymbol{w}}(C) \geq \rho$ and $\boldsymbol{w}_\rho(C) \geq k\rho$.*

[13]When it is clear from the context, for brevity, we might say that e is a *heavy edge* instead of *ρ-heavy edge*.

Proof. The fact that $\mathbf{w}_\rho(C) \geq k\rho$ follows immediately from the definition of heavy edges. Let F_i be the set heaviest i edges in C with respect to $\mathbf{w}$. Since C contains at least k heavy edges, we have that for all $i < k$, $C \setminus F_i$ contains at least $k - i$ heavy edges. Therefore, we have $\mathsf{val}_\mathbf{w}(C) = \min_{i \leq k-1} \frac{\mathbf{w}(C \setminus F_i)}{k-i} \geq \min_{i \leq k-1} \frac{(k-i)\rho}{k-i} = \rho$. $\square$

Proposition 2.20 says that if a cut is not interesting it must be expensive as a normalized free cut (i.e., high $\mathsf{val}_\mathbf{w}(C)$) and as a graph cut (i.e., high $w_\rho(C)$). We next give a characterization that relates $\mathsf{val}_\mathbf{w}$ and the sizes of the cuts for interesting cuts.

Lemma 2.21. *Let C be an interesting cut. Then $\mathsf{val}_{\boldsymbol{w}}(C) \leq \mathsf{val}_{\boldsymbol{w}}(C, H_{\boldsymbol{w},\rho} \cap C) < \rho$ if and only if $\boldsymbol{w}_\rho(C) < k\rho$.*

Proof. $(\rightarrow)$ By definition of $\mathbf{w}_\rho$, we have

$$\mathbf{w}_\rho(C) = \mathbf{w}(C \setminus (H_{\mathbf{w},\rho} \cap C)) + \rho|H_{\mathbf{w},\rho} \cap C|. \tag{2.3}$$

If $\mathsf{val}_\mathbf{w}(C, H_{\mathbf{w},\rho} \cap C) < \rho$, then $\mathbf{w}(C \setminus H_{\mathbf{w},\rho} \cap C) < \rho(k - |H_{\mathbf{w},\rho} \cap C|)$. By Equation (2.3), we have $\mathbf{w}_\rho(C) < k\rho$.

$(\leftarrow)$ Denote $F = H_{\mathbf{w},\rho} \cap C$. By definition of val, we have

$$\mathsf{val}_\mathbf{w}(C) \leq \mathsf{val}_\mathbf{w}(C, F) = \frac{\mathbf{w}(C \setminus F)}{k - |F|} \overset{(2.3)}{=} \frac{\mathbf{w}_\rho(C) - \rho|F|}{k - |F|} < \frac{k\rho - \rho|F|}{k - |F|} = \rho.$$

$\square$

Proof of Theorem 2.19. For the first part, we begin by proving that $\mathsf{mincut}_{\mathbf{w}_\rho} < k\rho$. Let C^* be a cut such that $\mathsf{val}_\mathbf{w}(C^*) = \mathsf{OPT}_\mathbf{w}$. By Proposition 2.20, C^* must be interesting. Since $\mathsf{val}_\mathbf{w}(C^*) = \mathsf{OPT}_\mathbf{w} < (1+\gamma)\lambda = \rho$, Lemma 2.21 implies that we have $\mathbf{w}_\rho(C^*) < k\rho$. Therefore, $\mathsf{mincut}_{\mathbf{w}_\rho} < k\rho$.

Next, we prove that $\mathsf{mincut}_{\mathbf{w}_\rho} \geq k\rho/(1+\gamma)$. Let C be a cut, and denote $F = H_{\mathbf{w},\rho} \cap C$. If C is not interesting, then Proposition 2.20 implies that $\mathbf{w}_\rho(C) \geq k\rho \geq k\rho/(1+\gamma)$. If C is interesting, by definition of $\mathbf{w}_\rho$, we have

$$\mathbf{w}_\rho(C) = \mathbf{w}(C \setminus F) + \rho|F| \geq \mathsf{OPT}_{\mathbf{w}}(k - |F|) + \frac{\rho}{1+\gamma}|F| \geq \frac{\rho k}{1+\gamma}.$$

The last inequality follows since by assumption $\mathsf{OPT}_{\mathbf{w}} \geq \rho/(1+\gamma)$.

For the second part of the theorem, as $\mathbf{w}_\rho(C) < k\rho$, Proposition 2.20 implies that C is interesting. By Lemma 2.21, $\mathsf{val}_{\mathbf{w}}(C, H_{\mathbf{w},\rho} \cap C) < \rho = (1+\gamma)\lambda$. $\square$

2.4. Fast Approximate LP Solver

In this section, we construct the fast range punisher for the normalized free cut problem. Our algorithm cannot afford to maintain the actual MWU weights, so it will instead keep track of *lazy weights*. From now on, we will use $\mathbf{w}^{\text{mwu}}$ to denote the actual MWU weights and $\mathbf{w}$ the weights that our data structure maintains.

Theorem 2.22 (Fast Range Punisher). *Given graph G initial weight function $\boldsymbol{w}^{\text{init}}$ and two real values $\lambda, \varepsilon > 0$ such that $\lambda \leq \mathsf{OPT}_{\boldsymbol{w}^{\text{init}}}$, there is a randomized algorithm that iteratively applies* PunishMin *until the optimal with respect to the final weight function $\boldsymbol{w}^{\text{mwu}}$ becomes at least $\mathsf{OPT}_{\boldsymbol{w}^{\text{mwu}}} \geq (1+\varepsilon)\lambda$, in time $\widetilde{O}(|E| + K + \frac{1}{\varepsilon}\sum_{e \in E} \log(\cdot\frac{\boldsymbol{w}^{\text{mwu}}(e)}{\boldsymbol{w}^{\text{init}}(e)}))$, where K is the number of cuts punished.*

The following theorem is almost standard: the fast range punisher, together with a fast algorithm for approximating OPT_w for any weight $\mathbf{w}$, implies a fast approximate LP solver (e.g., see [CQ17a, Fle04]). For completeness, we provide the proof in the Appendix.

Theorem 2.23 (Fast LP Solver). *Given a fast range punisher as described in Theorem 2.22, and a near-linear time algorithm for approximating OPT_w for any weight function w, there is an algorithm that output $(1+O(\varepsilon))$-approximate solution to kECSS LP in $\tilde{O}(m/\varepsilon^2)$ time.*

Notice that the above theorem implies our main result, Theorem 2.1. The rest of this section is devoted to proving Theorem 2.22. Following the high-level idea of [CQ17a], our data structure has two main components:

- **Range cut-listing data structure**: This data structure maintains dynamic (truncated) weighted graph $(G, \mathbf{w}_\rho)$ and is able to find a (short description of) $(1+O(\varepsilon))$-approximate cut whenever one exists, that is, it returns a cut of size between λ and $(1+O(\varepsilon))\lambda$ for some parameter λ. Since our weight function $\mathbf{w}$ changes over time, the data structure also has an interface that allows such changes to be implemented. The data structure can be taken and used directly in a blackbox manner, thanks to [CQ17a].
- **Lazy weight data structures**: Notice that a fast range punisher can only afford the running time of $\widetilde{O}\left(\sum_e \log \frac{\mathbf{w}^{\text{mwu}}(e)}{\mathbf{w}^{\text{init}}(e)}\right)$ for updating weights, while in the MWU framework, some edges would have to be updated much more often. We follow the idea of [CQ17a] to maintain approximate (lazy) weights that do not get updated too often but are still sufficiently close to the real weights. We remark that $\mathbf{w}^{\text{mwu}}$ only depends on the sequence of cuts, that PunishMin actually punishes. This lazy weight data structure is responsible for maintaining $\mathbf{w}$ that satisfies the following invariant:

  Invariant 1. *We have* $(1-\varepsilon)\boldsymbol{w}^{\text{mwu}} \le \boldsymbol{w} \le \boldsymbol{w}^{\text{mwu}}$.

  That is, we allow $\mathbf{w}$ to underestimate weights, but they cannot deviate more than by a factor of $(1-\varepsilon)$. In this way, our data structure only needs to update the weight implicitly and output necessary increments to the cut listing data structure whenever the invariant is violated.

In sum, our range punisher data structures deal with three weight functions $\mathbf{w}$ (lazy weights), $\mathbf{w}_\rho$ (truncated lazy weights, used by the range cut listing data structure) and $\mathbf{w}^{\mathrm{mwu}}$ (actual MWU weights, maintained implicitly).

The rest of this section is organized as follows. In Section 2.4.1–Section 2.4.3, we explain the components that will be used in our data structure, and in Section 2.4.4, we prove Theorem 2.22 using these components.

2.4.1. Compact representation of cuts

This part serves as a "communication language" for various components in our data structure. Since a cut can have up to $\Omega(m)$ edges, the data structure cannot afford to describe it explicitly. We will use a compact representation of cuts [CQ17a], which allows us to describe any $(1+\varepsilon)$-approximate solution in a given weighted graph using $\tilde{O}(1)$ bits; notice that, in the MWU framework, we only care about (punishing) near-optimal solutions, so it is sufficient for us that we are able to concisely describe such cuts.

Formally, we say that a family $\mathcal{F}$ of subsets of edges is ε-canonical for $(G, \mathbf{w})$ if (i) $|\mathcal{F}| \leq \widetilde{O}(|E|)$, (ii) any $(1+\varepsilon)$-approximate minimum cut of $(G, \mathbf{w})$ is a disjoint union of at most $\tilde{O}(1)$ sets in $\mathcal{F}$, (iii) any set $S \in \mathcal{F}$ can be described concisely by $\tilde{O}(1)$ bits, and (iv) every edge in the graph belongs to $\tilde{O}(1)$ sets in $\mathcal{F}$. It follows that any $(1+\varepsilon)$-approximate cut admits a short description. Denote by $[[S]]$ a short description of cut $S \in \mathcal{F}$, and for each $(1+\varepsilon)$ approximate cut C, $[[C]]$ a short description of C.

Lemma 2.24 (implicit in [CQ17a])**.** *There exists a randomized data structure that, on input $(G, \boldsymbol{w})$, can be initialized in near-linear time, (w.h.p) constructs an ε-canonical family $\mathcal{F} \subseteq 2^{E(G)}$, and handles the following queries:*

- *Given a description $[[C]]$ of a $(1+\varepsilon)$-approximate cut, output a list of $\tilde{O}(1)$ subsets in $\mathcal{F}$ such that C is a disjoint union of those subsets in $\tilde{O}(1)$ time.*
- *Given a description of $[[S]]$, $S \in \mathcal{F}$, output a list of edges in S in $\widetilde{O}(|S|)$ time.*

2.4.2. Range Cut-listing Data Structure

The cut listing data structure is encapsulated in the following theorem.

Theorem 2.25 (Range Cut-listing Data Structure [CQ17a])**.** *The cut-listing data structure, denoted by $\mathcal{D}$, maintains dynamically changing weighted graph $(G, \widehat{\boldsymbol{w}})$ and supports the following operations.*

- $\mathcal{D}$.Init$(G, \boldsymbol{w}^{\text{init}}, \lambda, \varepsilon)$ *where G is a graph, $\widehat{\boldsymbol{w}}$ is an initial weight function, and $\mathsf{mincut}_{\widehat{\boldsymbol{w}}} \geq \lambda$: initialize the data structure and the weight $\widehat{\boldsymbol{w}} \leftarrow \boldsymbol{w}^{\text{init}}$ in $\tilde{O}(m)$ time.*
- $\mathcal{D}$.FindCut$()$: *output either a short description of a $(1+O(\varepsilon))$-approximate mincut $[[C]]$ or $\emptyset$ (when $\mathsf{mincut}_{\widehat{\boldsymbol{w}}} > (1+\varepsilon)\lambda$). The operation takes amortized $\tilde{O}(1)$ time.*
- $\mathcal{D}$.Increment(Δ) *where $\Delta = \{(e, \delta_e)\}$ is the set of increments (defined by a pair of an edge $e \in E$ and a value $\delta_e \in \mathbb{R}_{\geq 0}$): For each $(e, \delta_e) \in \Delta$, $\widehat{\boldsymbol{w}}(e) \leftarrow \widehat{\boldsymbol{w}}(e) + \delta_e$. The operation takes $\tilde{O}(|\Delta|)$ time (note that $|\Delta|$ corresponds to the number of increments).*

As outlined earlier, the cut listing data structure will be invoked with $\widehat{\mathbf{w}} = \mathbf{w}_\rho$.

2.4.3. Truncated Lazy MWU Increment

The data structure is formally summarized by the definition below.

Definition 2.26 (Truncated Lazy MWU Increment)**.** *A truncated lazy MWU increment denoted by $\mathcal{L}$ maintains the approximate weight function $\boldsymbol{w}$ explicitly, and exact weight $\boldsymbol{w}^{\text{mwu}}$ implicitly and supports the following operations:*[14]

- $\mathcal{L}$.Init$(G, \boldsymbol{w}^{\text{init}}, \rho)$ *where G is a graph, $\boldsymbol{w}^{\text{init}}$ is the initial weight function, $\rho \in \mathbb{R}_{>0}$: Intialize the data structure, and set $\boldsymbol{w} \leftarrow \boldsymbol{w}^{\text{init}}$.*

[14]This is implicit in the sense that w is divided into parts and they are internally stored in different memory segments. Whenever needed, the real weight can be constructed from the memory content in near-linear time.

- $\mathcal{L}$.Punish($[[C]]$) *where C is a cut: Internally punish the free cut (C, F) for some F (to be made precise later) and output a list of increment $\Delta = \{(e, \delta_e)\}$ so that for each $e \in E$, $\boldsymbol{w}^{\text{init}}(e)$ plus the total increment over e is $\boldsymbol{w}_\rho(e)$.*
- $\mathcal{L}$.Flush()*: Return the exact weight $\boldsymbol{w}^{\text{mwu}}$.*

Remark that the output list of increments returned by Punish is mainly for the purpose of syncing with the cut listing data structure (so it aims at maintaining $\mathbf{w}_\rho$ instead of $\mathbf{w}$). Also, in the Punish operation, the data structure must compute the set $F \subseteq C$ of free edges efficiently (these are the edges whose weights would not be increased). This is one of the reasons for which we cannot use the lazy update data structure in [CQ17a] as a blackbox. Appendix 2.6 will be devoted to proving the following theorem.

Theorem 2.27. *There exists a lazy MWU increment with the following time complexity: (i) init operation takes $\tilde{O}(m)$ time, (ii)* Punish *takes $\tilde{O}(K) + \widetilde{O}\left(\sum_e \log \frac{\boldsymbol{w}^{\text{mwu}}(e)}{\boldsymbol{w}^{\text{init}}(e)}\right)$ time in total where K is the number of calls to* Punish *and outputs at most $\widetilde{O}\left(\sum_e \log \frac{\boldsymbol{w}^{\text{mwu}}(e)}{\boldsymbol{w}^{\text{init}}(e)}\right)$ increments, and (iii) flush takes $\tilde{O}(m)$ time. Moreover, Invariant 1 is maintained throughout the execution.*

2.4.4. A Fast Range Punisher for Normalized Free Cut Problem

Now we have all necessary ingredients to prove Theorem 2.22. The algorithm is very simple and described in Algorithm 1. We initialize the cut-listing data structure $\mathcal{D}$ so that it maintains the truncated weight $\mathbf{w}_\rho$ and the lazy weight data structure $\mathcal{L}$. We iteratively use $\mathcal{D}$ to find a cheap cut in $(G, \mathbf{w}_\rho)$ until no such cut exists. Due to our mapping theorem, such a cut found can be used for our problem, and the data structure $\mathcal{L}$ is responsible for punishing the weights (Line 8) and returns the list of edges to be updated (this is for the cut-listing $\mathcal{D}$ to maintain its weight function $\mathbf{w}_\rho$).

Algorithm 1: FastRangePunisher$(G, \mathbf{w}, \lambda)$

```
Input  : $G, \mathbf{w}^{\text{init}}, \lambda, \varepsilon$ such that $\mathsf{OPT}_{\mathbf{w}^{\text{init}}} \geq \lambda$.
Output: a correct weight function $\mathbf{w} = \mathbf{w}^{\text{mwu}}$ such that $\mathsf{OPT}_{\mathbf{w}} \geq (1+\varepsilon)\lambda$.
1  $\mathbf{w} \leftarrow \mathbf{w}^{\text{init}}$ and $\rho \leftarrow (1+\varepsilon)\lambda$
2  Let $\mathbf{w}_\rho$ be the truncated weight function of $\mathbf{w}$.
3  if $\mathsf{mincut}_{\mathbf{w}_\rho} \geq k\rho$ then return $\boldsymbol{w}$.
4  Let $\mathcal{D}$ and $\mathcal{L}$ be cut listing data structure, and truncated lazy MWU increment.
5  $\mathcal{D}$.Init$(G, \mathbf{w}_\rho, k\rho/(1+\varepsilon), \varepsilon)$
6  $\mathcal{L}$.Init$(G, \mathbf{w}, \rho, \varepsilon)$
7  while $\mathcal{D}$.FindCut() returns $[[C]]$ do
8      $\Delta \leftarrow \mathcal{L}$.Punish$([[C]])$
9      $\mathcal{D}$.Increment$(\Delta)$
10 $\mathbf{w} \leftarrow \mathcal{L}$.Flush()
11 return $\boldsymbol{w}$.
```

Analysis. By input assumption, we have $\mathsf{OPT}_w \geq \lambda$. If $\mathbf{w}$ is returned at line 3, then $\mathsf{mincut}_{\mathbf{w}_\rho} \geq k\rho$. By Theorem 2.19(1), $\mathsf{OPT}_{\mathbf{w}^{\text{mwu}}} \geq \mathsf{OPT}_{\mathbf{w}} \geq \rho = (1+\varepsilon)\lambda$, and we are done (since minimum cut can be computed in near-linear time). Now, we assume that $\mathbf{w}$ is returned at the last line. The following three claims imply Theorem 2.22.

Claim 2.28. *For every cut $[[C]]$ returned by the range cut listing data structure during the execution of Algorithm 1, we have that $(C, H_{\mathbf{w},\rho} \cap C)$ is a $(1+O(\varepsilon))$-approximation to $\mathsf{OPT}_{\boldsymbol{w}^{\text{mwu}}}$ at the time $[[C]]$ is returned.*

We remark that it is important that our cut punished must be approximately optimal w.r.t. the actual MWU weight.

Proof. By definition of $\mathcal{L}$.Flush() operation, we always have that the exact weight function and approximate weight function are identical at the beginning of the loop. By definition of $\mathcal{L}$.Punish$([[C]])$, the total increment plus the initial weight at the beginning of the loop for every edge e is $\mathbf{w}_\rho(e)$ and Invariant 1 holds. Therefore, by definition of $\mathcal{D}$.Increment(Δ), the range cut-listing data structure maintains the weight function $\mathbf{w}_\rho$ internally. We now bound the approximation of each cut $[[C]]$ that $\mathcal{D}$.FindCut() returned. Let $F = H_{\mathbf{w},\rho} \cap C$. By definition of FindCut(), we have that $\mathbf{w}_\rho(C) < k\rho$. By Theorem 2.19(2),

$\mathsf{val}_{\mathbf{w}}(C,F) < (1+\varepsilon)\lambda$. By Invariant 1, we have that $\mathsf{val}_{\mathbf{w}^{\text{mwu}}}(C,\tilde{F}) < (1+O(\varepsilon))\lambda$. Since $\mathsf{OPT}_{\mathbf{w}^{\text{mwu}}} \geq \mathsf{OPT}_{w^{\text{init}}} \geq \lambda$, we have (C,F) is a $(1+O(\varepsilon))$-approximation to OPT_{w}. $\square$

Claim 2.29. *At the end of Algorithm 1, we have* $\mathsf{OPT}_{\boldsymbol{w}^{\text{mwu}}} \geq (1+\varepsilon)\lambda$.

Proof. Consider the time when $\mathcal{D}$.FindCut() outputs $\emptyset$. The fact that this procedure terminates means that $\mathsf{mincut}_{\mathbf{w}_\rho} \geq k\rho$. Therefore, Theorem 2.19(1) implies that $\mathsf{OPT}_{\mathbf{w}} \geq (1+\varepsilon)\lambda$. Let (C^*,F^*) be an optimal normalized free cut with respect to $\mathbf{w}^{\text{mwu}}$. We have

$$\begin{aligned}\mathsf{OPT}_{\mathbf{w}^{\text{mwu}}} &= \mathsf{val}_{\mathbf{w}^{\text{mwu}}}(C^*,F^*)\\ &\geq \mathsf{val}_{\mathbf{w}}(C^*,F^*)\\ &\geq \mathsf{OPT}_{\mathbf{w}}\\ &\geq (1+\varepsilon)\lambda\end{aligned}$$

where the first inequality follows from Invariant 1. $\square$

Claim 2.30. *Algorithm 1 terminates in* $\tilde{O}(m+K+\frac{1}{\varepsilon}\cdot\sum_{e\in E}\log(\frac{\boldsymbol{w}(e)}{\boldsymbol{w}^{\text{init}}(e)}))$ *time where K is the number of* Punish *operations.*

Proof. We first bound the running time due to truncated lazy MWU increment. By Theorem 2.27, the total running time due to $\mathcal{L}$ (i.e., $\mathcal{L}$.Init, $\mathcal{L}$.Punish, $\mathcal{L}$.Flush) is $\tilde{O}(m+K+\frac{1}{\varepsilon}\cdot\sum_{e\in E}\log(\frac{\mathbf{w}(e)}{\mathbf{w}^{\text{init}}(e)}))$ time where K is the number of Punish operations. We bound the running time due to cut-listing data structure. Observe that the number of cuts listed equals the number of calls of Punish operations, and the total number of edge increments in $\mathcal{D}$ is $\widetilde{O}\left(\frac{1}{\varepsilon}\cdot\sum_{e\in E}\log(\frac{\mathbf{w}(e)}{\mathbf{w}^{\text{init}}(e)})\right)$. By Theorem 2.25, the total running time due to $\mathcal{D}$ (i.e, $\mathcal{D}$.Init, $\mathcal{D}$.FindCut(), $\mathcal{D}$.Increment(Δ)) is as desired. $\square$

2.5. LP Rounding for kECSS (Proof of Theorem 2.15)

In this section, we show how to round the LP solution x found by invoking Theorem 2.1. The main idea is use a sampling technique to sparsify the support of x. On the subgraph $G' \subseteq G$ based on this sparsified support, we apply the 2-approximation algorithm of Khuller and Vishkin [KV94] to obtain a $(2+\varepsilon)$-approximation solution.

Let G be a graph with capacities c (we omit capacities whenever it is clear from the context). Our algorithm performs the following steps.

Step 1: Sparsification. We will be dealing with the following LP relaxation for kECSS.

$$\min\{\sum_{e\in E(G)} c(e)x_e : \sum_{e\in C\setminus S} x_e \geq k-|S|, \quad \forall C\in\mathcal{C}\ \forall S\in\{F:|F|\leq k-1\wedge F\subseteq C\}, x\geq 0\}$$

Denote by $\mathsf{LP}_{kECSS}(G)$ the optimal LP value on input G. We prove the following lemma in Appendix A.1 that will allow us to sparsify our graph without changing the optimal fractional value by too much:

Lemma 2.31. *Given an instance (G,c), and in $\widetilde{O}(m/\varepsilon^2)$ time, we can compute a subgraph G' having at most $\widetilde{O}(nk/\varepsilon^2)$ edges such that $\mathsf{LP}_{kECSS}(G') = (1\pm O(\varepsilon))\mathsf{LP}_{kECSS}(G)$.*

The first step is simply to apply this lemma to obtain G' from G.

Step 2: Reduction to k-arborescences. Next, we reduce the kECSS problem to the minimum-cost k-arborescence problem which, on capacitated directed graph (H,c_H), can be described as the following IP:

$$\min\{\sum_{e\in E(H)} c_H(e)z_e : \sum_{e\in\delta^+(C)} z_e \geq k \text{ for } C\in\mathcal{C}; z\in\{0,1\}^{E(H)}\}$$

where $\mathcal{C}$ is the set of all cuts C such that $\{r\} \subseteq C \subsetneq V(G)$. Denote by $\mathsf{OPT}_{ar}(H)$ and $\mathsf{LP}_{ar}(H)$ the optimal integral and fractional values[15] of the minimum-cost k-arborescence problem respectively. We use the following integrality of its polytope:

Theorem 2.32 ([Sch03], Corollary 53.6a)**.** *The minimum-cost k-arborescence's polytope is integral, so we have that $\mathsf{OPT}_{ar}(H) = \mathsf{LP}_{ar}(H)$ for every capacitated input graph H.*

For any undirected graph G, denote by $D[G]$ the directed graph obtained by creating, for each (undirected) edge uv in G, two edges $(u \to v)$ and $(v \to u)$ in $D[G]$ whose capacities are just $c(uv)$. We will use the following theorem by Khuller and Vishkin (slightly modified) that relates the optimal values of the two optimization problems.

Theorem 2.33. *For any graph (H, c), the following properties hold:*

- $\mathsf{LP}_{ar}(D[H]) \leq 2\mathsf{LP}_{kECSS}(H)$, *and*
- *Any feasible solution for k-arborescences in $D[H]$ induces a feasible kECSS solution in H of at most the same cost.*

Proof. For the first part of the theorem, let x denote the optimal solution in the relaxed LP of kECSS of graph H. We create a fractional solution z in $D[H]$ as follows: for every edge $e \in E$ in H, if e_1 and e_2 are the two opposite directed edges in $D[H]$ derived from e, we set $z_{e_1} = z_{e_2} = x_e$. It is clear that $c(z) = 2c(x)$. We just need to argue that z is feasible in the relaxed k-arboresences problem. Consider a cut $C \in \mathcal{C}$ (where $r \in C$ and $C \neq V$). As $x(C) \geq k$, $\sum_{e \in \delta^+(C)} z_e \geq k$. Furthermore, by Lemma 2.17, x satisfies the boxing constraint, that is, $0 \leq x_e \leq 1$ for all edges $e \in H$, implying that $0 \leq z_e \leq 1$ for all directed edges $e \in D[H]$. This shows that z is feasible and the first part of the theorem is proved.

For the second part, consider a feasible solution for k-arborescences in $D[H]$. If any of the two opposite directed edges is part of the k-arborescences, we include its corresponding

[15]The relaxation is simply $\min\{\sum_{e \in E(H)} c_H(e) z_e : \sum_{e \in \delta^+(C)} z_e \geq k$ for $C \in \mathcal{C}; z \in [0,1]^{E(H)}\}$

undirected edge in H as part of our induced solution. Clearly, the cost of the induced solution cannot be higher and it is a feasible kECSS solution, since it guarantees that the cut value is at least k for all cuts. $\square$

Note that Theorem 2.32 and the algorithm by Khuller and Vishkin imply that the integrality gap of the kECSS LP is at most 2. While this result is immediate, it seems to be a folklore. To the best of our knowledge, it was not explicitly stated anywhere in the literature. This integrality gap allows us to obtain the first part of Corollary 2.2. Our final tool to obtain Theorem 2.15 (and the second part of Corollary 2.2) is Gabow's algorithm:

Theorem 2.34 ([Gab95])**.** *Given a graph $G = (V, E, c)$ with positive cost function c, a fixed root $r \in V$, and let $c_{\max}$ be the maximum cost on edges, there exists an algorithm that in $\tilde{O}(km\sqrt{n}\log(nc_{\max}))$ time outputs the integral minimum-cost k-arborescence.*

Algorithm of Theorem 2.15. Now, using the graph G' created in the first step, we create $D[G']$, and invoke Gabow's algorithm to compute an optimal k-arborescence in $D[G']$. Let $S \subseteq E(G)$ be the induced kECSS solution.

The cost of S is at most:

$$\begin{aligned}\mathsf{OPT}_{ar}(D[G']) &\leq \mathsf{LP}_{ar}(D[G'])\\ &\leq 2\mathsf{LP}_{kECSS}(G')\\ &\leq 2(1+O(\varepsilon))\mathsf{LP}_{kECSS}(G)\\ &\leq 2(1+O(\varepsilon))\mathsf{OPT}_{kECSS}(G)\end{aligned}$$

The first inequality is due to Theorem 2.32. The second one is due to Theorem 2.33 (first bullet). The third one is due to Lemma 2.31.

Analysis. Step 1 takes $\tilde{O}(m/\varepsilon^2)$ time, by Lemma 2.31. As the sparsified G' has $m' = \tilde{O}(\frac{nk}{\varepsilon^2})$ edges, for Step 2, by Theorem 2.34, we can compute the arborescence in $O(km'\sqrt{n}\log(nc_{\max})) = \tilde{O}(\frac{k^2n^{1.5}}{\varepsilon^2}\log c_{\max})$ time. To remove the term $\log c_{\max}$ in our case, we can do as follows.

Polynomially bounded costs. Since Gabow's algorithm for arborescences has the running time depending on $c_{\max}$, the maximum cost of the edges, we discuss here how to ensure that $c_{\max}$ is polynomially bounded.

Let x be the LP solution obtained from our LP solver. Denote by $C^* = \sum_{e\in E} c_e x_e$, so we have that C^* is between $\mathsf{OPT}/2$ and OPT, where OPT is the optimal integral value.

First, whenever we see an edge $e \in E$ with $c_e > 2C^*$, we remove such an edge from the graph G. For each remaining edge $e \in E$, we round the capacity c_e up to the next multiple of $M = \lceil \varepsilon C^*/|E| \rceil$. So, after this rounding up, we have the capacities in $\{M, 2M, \ldots, C^*\}$, and we can then scale them down by a factor of M so that the resulting capacities c'_e are between 1 and $O(|E|/\varepsilon)$. It is an easy exercise to verify that any α-approximation algorithm for (G, c') can be turned into an $\alpha(1+\varepsilon)$-approximation algorithm for (G, c).

In summary, the total running time is $\tilde{O}\left(\frac{m}{\varepsilon^2} + \frac{k^2n^{1.5}}{\varepsilon^2}\right)$. Notice that the running time can be $\tilde{O}(\frac{m}{\varepsilon^2} + T_k(kn/\varepsilon^2, n))$ if we let the running time of Theorem 2.34 be $T_k(m, n)$, this complete the proof for Theorem 2.15.

2.6. Truncated Lazy MWU Increment (Proof of Theorem 2.27)

2.6.1. Additive Accuracy

Notice that, at any time, we maintain $\mathbf{w}^{\mathrm{mwu}}(e) = \frac{1}{c(e)} \cdot \exp\left(\mathbf{v}^{\mathrm{mwu}}(e)s(e)\right)$ by some positive real numbers $\mathbf{v}^{\mathrm{mwu}}(e)$ and $s(e) = \frac{\varepsilon}{c(e)}$. For the true vector $\mathbf{v}^{\mathrm{mwu}}$, the update rule for (C, F) becomes the following: $\mathbf{v}^{\mathrm{mwu}}(e) \leftarrow \mathbf{v}^{\mathrm{mwu}}(e) + c_{\min}$ for all $e \in C \setminus F$. This update causes all edges in $C \setminus F$ to increase their $\mathbf{v}^{\mathrm{mwu}}(e)$ by the same amount of $c_{\min}$. By Theorem 2.19(2), F can be assumed to be $H_{\rho,\mathbf{w}} \cap C$, i.e., the set of heavy edges with respect to $\mathbf{w}$ inside C. From now on, we always use $H_{\mathbf{w},\rho} \cap C$ as a free edge set whenever we punish C.

Instead of maintaining the approximate vector $\mathbf{w}$ for the real vector $\mathbf{w}^{\mathrm{mwu}}$, we instead work with the additive form of the approximate vector $\mathbf{v}$ for the real vector $\mathbf{v}^{\mathrm{mwu}}$, and we bound the additive error:

$$\forall e \in E, \mathbf{v}^{\mathrm{mwu}}(e) - \eta/s(e) \leq \mathbf{v}(e) \leq \mathbf{v}^{\mathrm{mwu}}(e) \tag{2.4}$$

Next, we show that it is enough to work on $\mathbf{v}^{\mathrm{mwu}}$ with additive errors.

Proposition 2.35. *If Equation (2.4) holds, then* $\forall e \in E, \boldsymbol{w}^{\mathrm{mwu}}(e)(1-\eta) \leq \boldsymbol{w}(e) \leq \boldsymbol{w}^{\mathrm{mwu}}(e)$.

Proof. Fix an arbitrary edge $e \in E$, we have $\mathbf{w}(e) \leq \mathbf{w}^{\mathrm{mwu}}(e)$. Moreover,

$$\mathbf{w}(e) \geq \frac{1}{c(e)}\exp((\mathbf{v}^{\mathrm{mwu}}(e) - \eta/s(e))s(e)) = \frac{1}{c(e)} \cdot \frac{\exp(\mathbf{v}^{\mathrm{mwu}}(e)s(e))}{\exp(\eta)} \geq (1-\eta)\mathbf{w}^{\mathrm{mwu}}(e).$$

□

2.6.2. Local Bookkeeping

We describe the set of variables to maintain in order to support PUNISH operation efficiently. Let $\mathcal{F}$ be a ε-canonical family of subsets of edges (as defined in Lemma 2.24). We call each subset of edges in $\mathcal{F}$ as a canonical cut. Let $\bar{E} = E \setminus H_{\mathbf{w},\rho}$ be the set of non-heavy edges where $H_{\rho,\mathbf{w}} = \{e \in E \colon \mathbf{w}(e) \geq \rho\}$ is the set of heavy edges. We define a bipartite graph $\mathcal{B} = (\mathcal{F}, \bar{E}, E_{\mathcal{B}})$ where the first vertex partition is the set of canonical cuts $\mathcal{F}$, the second vertex partition is $\bar{E}$, and for each $S \in \mathcal{F}$ and for each $e \in \bar{E}$, we add an edge (S, e) to $E_{\mathcal{B}}$ if and only if $e \in S$. Let $\mathrm{q}(\mathcal{B}) =$ the maximum degree of nodes in $\bar{E}$ in graph $\mathcal{B}$. Since $\mathcal{F}$ is ε-canonical, $\mathrm{q}(\mathcal{B}) = \tilde{O}(1)$. By Lemma 2.24, given a description $[[C]]$ of 1 or 2$-$repsecting cut, we can compute a list of at most $\tilde{O}(1)$ canonical cuts in $\mathcal{F}$ in $\tilde{O}(1)$ time.

We maintain the following variables:

(1) For each canonical cut $S \in \mathcal{F}$,

(a) we have a non-negative real number $\mathrm{ref}(S)$ representing the reference point for the total increase in S so far.

(b) in addition, we create a min priority queue Q_S containing the set of neighbors $N_{\mathcal{B}}(S)$ (which is the set of edges in $\bar{E}$ that S contains).

(c) moreover, we define $c_{\mathcal{B}}(S) = \min_{e \in N_{\mathcal{B}}(S)} c(e)$ for the purpose of computing $c_{\min}$ which is the minimum capacity $c(e)$ for all edge e in the cut (excluding heavy edges) that we want to punish.

(2) For each edge $(S, e) \in E_{\mathcal{B}}$, we have a number $\mathrm{last}(S, e)$ representing the last update point for e in S.

(3) For each edge $e \in E$, we maintain $\mathbf{v}(e)$.

For each edge $(S, e) \in E_{\mathcal{B}}$, we define $\mathrm{diff}(S, e) = \mathrm{ref}(S) - \mathrm{last}(S, e) \geq 0$. This difference represents the total slack from the exact weight of e on S (we will ensure that the slack is non-negative by being "lazy"). When summing over all canonical cuts that contains e, we ensure that $\sum_{S \ni e} \mathrm{diff}(S, e) = \mathbf{v}^{\mathrm{mwu}}(e) - \mathbf{v}(e)$. More formally, we maintain the following invariants throughout the execution of the truncated lazy increment.

Invariant 2. *Let $\eta' = \eta / \mathrm{q}(\mathcal{B})$.*

(1) for all $e \in \bar{E}$, $\frac{\eta}{s(e)} \geq \sum_{S:(S,e) \in E_{\mathcal{B}}} \mathrm{diff}(S, e) = \boldsymbol{v}^{\mathrm{mwu}}(e) - \boldsymbol{v}(e)$,

(2) for all $(S, e) \in E_{\mathcal{B}}$, $Q_S.\mathrm{priority}(e) = \mathrm{last}(S, e) + \frac{\eta'}{s(e)}$, and

(3) for all $e \in H_{\boldsymbol{w},\rho}$, $\boldsymbol{v}^{\mathrm{mwu}}(e) = \boldsymbol{v}(e)$.

Intuitively, the first invariant means for each $e \in \bar{E}$, the total difference over all $S \ni e$ is bounded. The second invariant ensures that $\mathrm{ref}(S) \leq Q_S.\mathrm{priority}(e)$ if and only if $\mathrm{diff}(S, e)$ is small, and we can apply extract min operations on Q_S to detect all edges whose priority exceeds the reference point efficiently. The third invariant means we restore the exact value for all heavy edges.

Proposition 2.36. *Invariant 21 implies Equation (2.4).*

Algorithm 2: $\textsc{Reset}(e)$

```
1 $\mathbf{v}(e) \leftarrow \mathbf{v}(e) + \sum_{S:(S,e)\in E_{\mathcal{B}}} \mathrm{diff}(S,e)$
2 for each $S : (S,e) \in E_{\mathcal{B}}$ do
3     $\mathrm{last}(S,e) \leftarrow \mathrm{ref}(S)$
4     $Q_S.\mathrm{priority}(e) \leftarrow \mathrm{last}(S,e) + \frac{\eta'}{s(e)}$
```

Also, this invariant allows us to "reset" $\mathbf{v}$ to be $\mathbf{v}^{\mathrm{mwu}}$ efficiently.

Since priority queue supports the change of priority in $O(\log m)$ time, we have:

Proposition 2.37. *The procedure* $\textsc{Reset}$ *can be implemented in time* $O(\mathrm{q}(\mathcal{B}) \cdot \log m) = \tilde{O}(1)$.

2.6.3. Init

Define $\mathbf{v} = v_0$ where v_0 is the additive form of w_0. We construct the bipartite graph $\mathcal{B} = (\mathcal{F}, \bar{E}, E_{\mathcal{B}})$ as defined in Section 2.6.2. We use Lemma 2.24 to construct $\mathcal{B}$ in $\tilde{O}(m)$ time. For each $S \in \mathcal{F}$, we create a min priority queue Q_S containing all the elements in $N_{\mathcal{B}}(S)$ where for each $e \in N_{\mathcal{B}}(S)$, we set $Q_S.\mathrm{priority}(e) = \eta'/s(e)$. We also define $\mathrm{ref}(S) = 0$ for all $S \in \mathcal{F}$, and $\mathrm{last}(S,e) = 0$ for all $(S,e) \in E_{\mathcal{B}}$. By design, the invariants are satisfied. The total running time of this step is $\tilde{O}(m)$.

2.6.4. Punish

Given a short description of 1 or 2-respecting cut $[[C]]$, we apply Lemma 2.24 to obtain a set $\mathcal{S} \subseteq \mathcal{F}$ of $\tilde{O}(1)$ canonical cuts whose disjoint union is C in $\tilde{O}(1)$ time. Recall that the update increases $\mathbf{v}^{\mathrm{mwu}}(e)$ by $c_{\min}$ for each $e \in C - H_{\mathbf{w},\rho}$ where $c_{\min} = \min_{e\in C - H_{\mathbf{w},\rho}} c(e)$.

Claim 2.38. *We can compute* $c_{\min}$ *in* $\tilde{O}(1)$ *time.*

Proof. By definition of $c_{\mathcal{B}}(S)$, $\min_{S\in\mathcal{S}} c_{\mathcal{B}}(S) = \min_{S\in\mathcal{S}} \min_{e\in N_{\mathcal{B}}(S)} c(e) = \min_{e\in C-H_{\mathbf{w},\rho}} c(e) = c_{\min}$. The claim follows because there are $\tilde{O}(1)$ canonical cuts in $\mathcal{S}$ and we maintain the value $c_{\mathcal{B}}(S)$ for every $S \in \mathcal{F}$. □

In the first step, for each $S \in \mathcal{S}$, we set $\mathrm{ref}(S) \leftarrow \mathrm{ref}(S) + c_{\min}$. This takes $\tilde{O}(1)$ time because $|\mathcal{S}| = \tilde{O}(1)$ and potentially causes a violation to Invariant 21.

In the second step, we check and fix the invariant violation as follows. For each $S \in \mathcal{S}$, let $W_S = \{e \in S \setminus H_{\mathbf{w},\rho} \colon \mathrm{ref}(S) > Q_S.\,\mathrm{priority}(e)\}$ be the set of all edges in $S \setminus H_{\mathbf{w},\rho}$ whose priority in Q_S is smaller than the reference point $\mathrm{ref}(S)$. For each $e \in W_S$, we call the procedure Reset(e). This take times $O(r \cdot q(\mathcal{B}) \log m) = \tilde{O}(r)$ where r is the number of calls to Reset procedure. There will be new heavy edges after this step, which means that we need to update $\mathcal{B}$ to correct the set $\bar{E}$.

In the third step, we identify new heavy edges from the set of edges that we called Reset procedure in the second step, then we remove each edge in the set from the associated priority queues and from the graph $\mathcal{B}$ as follows. Let $U = \bigcup_{S \in \mathcal{S}} W_S$. Define $U_H = \{e \in U \colon \mathbf{w}(e) \geq \rho\}$. For each $e \in U_H$, for all $D \in N_{\mathcal{B}}(e)$, remove e from the priority queue Q_D and update the value $c_{\mathcal{B}}(D)$ (to get a new minimum after removing e). Finally, delete all nodes in U_H from $\mathcal{B}$. The third step takes $O(|U| + |U_H|\,\mathrm{q}(\mathcal{B}) \log m + |U_H|\,\mathrm{q}(\mathcal{B})) = \tilde{O}(r)$ time. The running time follows because the $|U| = r$ and $|U_H| \leq |U|$.

Finally, we output Δ where Δ is constructed as follows. For each $e \in U$, let $\mathbf{w}'(e)$ be the weight of e before Reset(e) is invoked. If $e \notin U_H$, then we define $\delta_e = \mathbf{w}(e) - \mathbf{w}'(e)$. Otherwise, we define $\delta_e = \rho - \mathbf{w}'(e)$. Then, we add (e, δ_e) to Δ.

Lemma 2.39. *If Invariant 2 holds before calling* Punish$([[C]])$, *then Invariant 2 holds afterwards.*

Proof. In the first step, we have $\bigcup_{S \ni \mathcal{S}} N_{\mathcal{B}}(S) = C \setminus H_{\mathbf{w},\rho}$, and thus the violation to Invariant 21 can only happen due to some edge $e \in C \setminus H_{\mathbf{w},\rho}$. Because the unions are over disjoint sets, for each edge $e \in C \setminus H_{\mathbf{w},\rho}$, there is a unique canonical cut $S_e \in \mathcal{S}$ such that $N_{\mathcal{B}}(S_e) \ni e$.

Claim 2.40. *If there is a violation to Invariant 21 due to an edge $e \in C \setminus H_{\boldsymbol{w},\rho}$, then* Reset$(e)$ *is invoked in the second step.*

Proof. Since Invariant 21 is violated due to an edge e, we have $\sum_{S':(S',e)\in E_{\mathcal{B}}} \mathrm{diff}(S',e) > \eta/s(e)$. By averaging argument, there is a canonical cut S^* such that $\mathrm{diff}(S^*,e) > \frac{\eta}{s(e)} \cdot q(\mathcal{B})$. Since $\mathrm{diff}(S_e,e)$ is the only term in the summation that is increased, we have $S^* = S_e$. Therefore, we have

$$\frac{\eta}{s(e)} \cdot q(\mathcal{B}) < \mathrm{diff}(S_e,e) = \mathrm{ref}(S_e) - \mathrm{last}(S_e,e) \overset{2}{=} \mathrm{ref}(S_e) - Q_{S_e}.\mathrm{priority}(e) + \frac{\eta'}{s(e)}.$$

Therefore, $\mathrm{ref}(S_e) > Q_{S_e}.\mathrm{priority}(e)$, and so $e \in W_S$ as defined in the second step. Hence, Reset(e) is invoked. $\square$

Since Reset(e) is invoked for every violation, we have that Invariant 21 is maintained. By design, the second invariant is trivially maintained whenever Reset is invoked, and also the last invariant is automatically maintained by the third step. This completes the proof. $\square$

2.6.5. Flush

For each $e \in \bar{E}$, we call the procedure Reset(e). Then, we output $\mathbf{w}$ which is the same as $\mathbf{w}^{\mathrm{mwu}}$. The total running time is $O(q(\mathcal{B})|\bar{E}|) = \tilde{O}(m)$.

2.6.6. Total Running Time

The initialization takes $\tilde{O}(m)$. Let K be the number of calls to Punish$([[C]])$ and let I be the number of calls to Reset(e) before calling Flush$()$. The total running time due to the first step is $O(K \log^2 n) = \tilde{O}(K)$, and total running time due to the second and third steps is $\tilde{O}(I)$. It remains to bound I, the total number of calls to Reset(e). Since each Reset(e) increases of weight $\mathbf{w}^{\mathrm{mwu}}(e)$ by a factor of $1+\eta'$, the total number of resets is

$$O(\sum_{i\in[n]} \log_{1+O(\eta')}\left(\frac{\mathbf{w}^{\text{mwu}}(e)}{\mathbf{w}^{\text{init}}(e)}\right)) = O\left(\frac{q(\mathcal{B})}{\eta}\cdot\sum_{i\in[n]}\log(\frac{\mathbf{w}^{\text{mwu}}(e)}{\mathbf{w}^{\text{init}}(e)})\right)$$
$$= \tilde{O}\left(\frac{1}{\eta}\cdot\sum_{i\in[n]}\log(\frac{\mathbf{w}^{\text{mwu}}(e)}{\mathbf{w}^{\text{init}}(e)})\right).$$

2.7. Authors

This chapter was written by Parinya Chalermsook, Chien-Chung Huang, Danupon Nanongkai, Thatchaphol Saranurak, Pattara Sukprasert, and Sorrachai Yingchareonthawornchai. It was published at The International Colloquium on Automata, Languages and Programming (ICALP) 2022 [CHN+22].

CHAPTER 3

Dynamic Spanners

Increasingly, algorithms are used interactively for data analysis, decision making, and classically as data structures. Often it is not realistic to assume that a user or an adversary is *oblivious* to the outputs of the algorithms; they can be *adaptive* in the sense that their updates and queries to the algorithm may depend on the previous outputs they saw. Unfortunately, many classical algorithms give strong guarantees only when assuming an oblivious adversary. This calls for the design of algorithms that work against an adaptive adversary whose performance match the ones assuming an oblivious adversary. Driven by this question, there have been exciting lines of work across different communities in theoretical computer science, including streaming algorithms against an adaptive adversary [BEJWY20, HKM+20, WZ20, ABED+21, KMNS21, BHM+21], statistical algorithms against an adaptive data analyst [HU14, DFH+15, BNS+21, SU17], and very recent algorithms for machine unlearning [GJN+21].

In the area of dynamic graph algorithms, a continuous effort has also been put on designing algorithms against an adaptive adversary. This is witnessed by dynamic algorithms for maintaining spanning forests [HDLT01, NS17b, Wul17, NSW17, CGL+19], shortest paths [BC16, Ber17, BC17, CK19, CS21, GWN20a, GWN20b, GWW20, Chu21], matching [BHI15, BHN16, BHN17, BK19, Waj20, BK21], and more. This development led to new powerful tools, such as the expander decomposition and hierarchy [SW19, GRST21, LS21] applicable beyond dynamic algorithms [Li21, LPS21, AKT21, Zha21], and other exciting applications such as the first almost-linear time algorithms for many flow and cut problems [vdBLN+20, BLL+21, Chu21, BGS21]. Nevertheless, for many fundamental dynamic graph problems, including graph sparsifiers [ADK+16], reachability [BPW19], directed shortest paths [GWN20a], the performance gap between algorithms against an oblivious and adaptive adversary remains large, waiting to be explored and, hopefully, closed.

One of the most prominent dynamic problems whose oblivious-vs-adaptive gap is maximally large is the *fully dynamic spanner* problem [AFI06, Elk11, BKS12, BK16, FG19, BFH19, BBG+20]. Given an unweighted undirected graph $G = (V, E)$ with n vertices, an α-*spanner* H is a subgraph of G whose pairwise distances between vertices are preserved up to the *stretch* factor of α, i.e., for all $u, v \in V$, we have $\text{dist}_G(u, v) \le \text{dist}_H(u, v) \le \alpha \cdot \text{dist}_G(u, v)$.[1] In this problem, we want to maintain an α-spanner of a graph G while G undergoes both edge insertions and deletions, and for each edge update, spend as small update time as possible.

Assuming an oblivious adversary, near-optimal algorithms have been shown: for every $k \ge 1$, there are algorithms maintaining a $(2k-1)$-spanner containing $\tilde{O}(n^{1+1/k})$ edges,[2] which is nearly tight with the $\Omega(n^{1+1/k})$ bound from Erdős' girth conjecture (proven for the cases where $k = 1, 2, 3, 5$ [Wen91]). Their update times are either $O(k \log^2 n)$ amortized [BKS12, FG19] or $O(1)^k \log^3 n$ worst-case [BFH19], both of which are polylogarithmic when k is a constant.

In contrast, the only known dynamic spanner algorithm against an adaptive adversary by [AFI06] requires $O(n)$ amortized update time and it can maintain a $(2k-1)$-spanner of size $O(n^{1+1/k})$ only for $k \le 3$. Whether the $O(n)$ bound can be improved remained open until very recently. Bernstein et al. [BBG+20] show that a $\log^6(n)$-spanner can be maintained against an adaptive adversary using polylog(n) amortized update time. The drawback, however, is that their expander-based technique is too crude to give any stretch smaller than polylog(n). Hence, for $k \le \log^6(n)$, it is still unclear if the $\Theta(n)$ bound is inherent. Surprisingly, this holds even if we allow infinite time, and only count *recourse*, i.e., the number of edge changes

[1]Here, $\text{dist}_G(u, v)$ denotes the distance between u and v in graph G.
[2]$\tilde{O}(\cdot)$ hides a polylog(n) factor.

per update in the maintained spanner. The stark difference in performance between the two adversarial settings motivates the main question of this chapter:

Is the $\Omega(n)$ recourse bound inherent for fully dynamic spanners against an adaptive adversary?

Recourse is an important performance measure of dynamic algorithms. There are dynamic settings where changes in solutions are costly while computation itself is considered cheap, and so the main goal is to directly minimize recourse [GKS14, GK14, ABL+20, GL20]. Even when the final goal is to minimize update time, there are many dynamic algorithms that crucially require the design of subroutines with recourse bounds *stronger than* update time bounds to obtain small final update time [CZ20, GRST21, CGH+20]. Historically, there are dynamic problems, such as planar embedding [HR20b, HR20a] and maximal independent set [CHK16, BDH+19, CZ19], where low recourse algorithms were first discovered and later led to fast update-time algorithms. Similar to dynamic spanners, there are other fundamental problems, including topological sorting [BC18] and edge coloring [BGW21], for which low recourse algorithms remain the crucial bottleneck to faster update time.

In this part, we successfully break the $O(n)$ recourse barrier and completely close the oblivious-vs-adaptive gap with respect to recourse for fully dynamic spanners against an adaptive adversary.

Theorem 3.1. *There exists a deterministic algorithm that, given an unweighted graph G with n vertices undergoing edge insertions and deletions and a parameter $k \geq 1$, maintains a $(2k-1)$-spanner of G containing $O(n^{1+1/k} \log n)$ edges using $O(\log n)$ amortized recourse.*

As the above algorithm is deterministic, it automatically works against an adaptive adversary. Each update can be processed in polynomial time. Both the recourse and stretch-size trade-off of Theorem 3.1 are optimal up to a $O(\log n)$ factor. When ignoring the update time, it even dominates the current best algorithm assuming an oblivious adversary [BKS12, FG19] that maintains a $(2k-1)$-spanner of size $O(n^{1+1/k}\log n)$ using $O(k\log^2 n)$ recourse. Therefore, the oblivious-vs-adaptive gap for amortized recourse is closed.

The algorithm of Theorem 3.1 is as simple as possible. As it turns out, a variant of the classical greedy spanner algorithm [ADD+93a] simply does the job! Although the argument is short and "obvious" in hindsight, for us, it was very surprising. This is because the greedy algorithm *sequentially* inspects edges in some fixed order, and its output solely depends on this order. Generally, long chains of dependencies in algorithms are notoriously hard to analyze in the dynamic setting. More recently, a similar greedy approach was dynamized in the context of dynamic maximal independent set [BDH+19] by choosing a *random order* for the greedy algorithm. Unfortunately, the random order must be kept secret from the adversary and so this fails completely in our adaptive setting. Despite these intuitive difficulties, our key insight is that we can *adaptively choose the order* for the greedy algorithm after each update. This simple idea is enough, see Section 3.1 for details.

Theorem 3.1 leaves open the oblivious-vs-adaptive gap for the update time. Below, we show a partial progress on this direction by showing an algorithm with near-optimal recourse and simultaneously non-trivial update time.

Theorem 3.2. *There exists a randomized algorithm that, given an unweighted graph G with n vertices undergoing edge insertions and deletions, with high probability maintains against an adaptive adversary a 3-spanner of G containing $\tilde{O}(n^{1.5})$ edges using $\tilde{O}(1)$ amortized recourse* and *$\tilde{O}(\sqrt{n})$ worst-case update time.*

We note again that, prior to the above result, there was no algorithm against an adaptive adversary with $o(n)$ *amortized update time* that can maintain a spanner of stretch less than $\log^6(n)$. Theorem 3.2 shows that for 3-spanners, the update time can be $\tilde{O}(\sqrt{n})$ worst-case, while guaranteeing near-optimal recourse.

We prove Theorem 3.2 by employing a technique called *proactive resampling*, which was recently introduced in [BBG+20] for handling an adaptive adversary. We apply this technique on a modification of a spanner construction by Grossman and Parter [GP17] from distributed computation community. The modification is small, but seems inherently necessary for bounding the recourse.

To successfully apply proactive resampling, we refine the technique in two ways. First, we present a simple abstraction in terms of a certain load balancing problem that captures the power of proactive resampling. Previously, the technique was presented and applied specifically for the dynamic cut sparsifier problem [BBG+20]. But actually, this technique is conceptually simple and quite generic, so our new abstraction will likely facilitate future applications. Our second technical contribution is to generalize and make the proactive resampling technique more flexible. At a very high level, in [BBG+20], there is a single parameter about sampling probability that is *fixed* throughout the whole process, and their analysis requires this fact. In our load-balancing abstraction, we need to work with multiple sampling probabilities and, moreover, they change through time. We manage to analyze this generalized process using probabilistic tools about *stochastic domination*, which in turn simplifies the whole analysis.

If a strong recourse bound is not needed, then proactive resampling can be bypassed and the algorithm becomes very simple, deterministic, and has slightly improved bounds as follows.

Theorem 3.3. *There exists a deterministic algorithm that, given an unweighted graph G with n vertices undergoing edge insertions and deletions, maintains a 3-spanner of G containing $O(n^{1.5})$ edges using $O(\min\{\Delta, \sqrt{n}\}\log n)$ worst-case update time, where Δ is the maximum degree of G.*

Ref.	**Stretch**	**Size**	**Recourse**	**Update Time**	**Deterministic?**
Against an oblivious adversary					
[BKS12]	$2k-1$	$O(k^8 n^{1+1/k}\log^2 n)$	$O(7^{k/2})$ amortized		rand. oblivious
[BKS12, FG19]	$2k-1$	$O(n^{1+1/k}\log n)$	$O(k\log^2 n)$ amortized		rand. oblivious
[BFH19]	$2k-1$	$\tilde{O}(n^{1+1/k})$	$O(1)^k\log^3 n$ worst-case		rand. oblivious
Against an adaptive adversary					
[AFI06]	3	$O(n^{1+1/2})$	$O(\Delta)$ amortized		deterministic
	5	$O(n^{1+1/3})$	$O(\Delta)$ amortized		deterministic
[BBG+20]	$\tilde{O}(1)$	$\tilde{O}(n)$	$\tilde{O}(1)$ amortized		rand. adaptive
	$n^{o(1)}$	$\tilde{O}(n)$	$n^{o(1)}$ worst-case		deterministic
Ours	$2k-1$	$O(n^{1+1/k}\log n)$	$O(\log n)$ amortized	poly(n) worst-case	deterministic
	3	$\tilde{O}(n^{1+1/2})$	$\tilde{O}(1)$ amortized	$\tilde{O}(\sqrt{n})$ worst-case	rand. adaptive
	3	$O(n^{1+1/2})$	$O(\min\{\Delta,\sqrt{n}\}\log n)$ worst-case		deterministic

Table 3.1. The state of the art of fully dynamic spanner algorithms.

Despite its simplicity, the above result improves the update time of the fastest deterministic dynamic 3-spanner algorithm [AFI06] from $O(\Delta)$ amortized to $O(\min\{\Delta,\sqrt{n}\}\log n)$ worst-case. In fact, all previous dynamic spanner algorithms with worst-case update time either assume an oblivious adversary [Elk11, BK16, BFH19] or have a very large stretch of $n^{o(1)}$ [BBG+20]. See Table 3.1 for detailed comparison.

Organization. In Section 3.1, we give a very short proof of Theorem 3.1. In Section 3.2, we prove Theorem 3.2 assuming a crucial lemma (Lemma 3.8) needed for bounding the recourse. To prove this lemma, we show a new abstraction for the proactive resampling technique in Section 3.3 and complete the analysis in Section 3.4. Our side result, Theorem 3.3, is based on the the static construction presented in Section 3.2.1 and its simple proof is given in Section 3.2.2.

3.1. Deterministic Spanner with Near-optimal Recourse

Below, we show a decremental algorithm that *handles edge deletions only* with near-optimal recourse. This will imply Theorem 3.1 by a known reduction formally stated in Lemma 3.6. To describe our decremental algorithm, let us first recall the classic greedy algorithm.

The Greedy Algorithm. Althöfer et al. [ADD$^+$93b] showed the following algorithm for computing $(2k-1)$-spanners. Given a graph $G = (V, E)$ with n vertices, fix some *order* of edges in E. Then, we inspect each edge one by one according to the order. Initially $E_H = \emptyset$. When we inspect $e = (u, v)$, if $\text{dist}_H(u, v) \geq 2k$, then add e into E_H. Otherwise, ignore it. We have the following theorem.

Theorem 3.4 ([ADD$^+$93b]). *The greedy algorithm above outputs a $(2k-1)$-spanner $H = (V, E_H)$ of G containing $O(n^{1+1/k})$ edges.*

It is widely believed that the greedy algorithm is extremely bad in dynamic setting as an edge update can drastically change the greedy spanner. In contrary, when we allow the order in which greedy scans the edges to be changed adaptively, we can avoid removing spanner edges until it is deleted by the adversary. This key insight leads to optimal recourse.

When recourse is the only concern, prior to our work this result was known only for spanners with polylog stretch, which is a much easier problem.

The Decremental Greedy Algorithm. Now we describe our deletion-only algorithm. Let G be an initial graph with m edges and $H = (V, E_H)$ be a $(2k-1)$-spanner with $O(n^{1+1/k})$ edges. Suppose an edge $e = (u, v)$ is deleted from the graph G. If $(u, v) \notin E_H$, then we do nothing. Otherwise, we do the following. We first remove e from E_H. Now we look at the only remaining non-spanner edges $E \setminus E_H$, one by one in an arbitrary order. (Note that the order is *adaptively* defined and not fixed through time because it is defined only on $E \setminus E_H$.) When we inspect $(x, y) \in E \setminus E_H$, as in the greedy algorithm, we ask if $\text{dist}_H(x, y) \geq 2k$ and add (x, y) to E_H if and only if it is the case. This completes the description of the algorithm.

Analysis. We start with the most crucial point. We claim that the new output after removing e is as if we run the greedy algorithm that first inspects edges in E_H (the order within E_H is preserved) and later inspects edges in $E \setminus E_H$.

To see the claim, we argue that if the greedy algorithm inspects E_H first, then the whole set E_H must be included, just like E_H remains in the new output. To see this, note that, for each $(x, y) \in E_H$, $\text{dist}_H(x, y) \geq 2k$ when (x, y) was inspected according to some order. But, if we move the whole set E_H to be the prefix of the order (while the order within E_H is preserved), it must still be the case that $\text{dist}_H(x, y) \geq 2k$ when (x, y) is inspected and so e must be added into the spanner by the greedy algorithm.

So our algorithm indeed "simulates" inspecting E_H first, and then it explicitly implements the greedy algorithm on the remaining part $E \setminus E_H$. So we conclude that it simulates the greedy algorithm. Therefore, by Theorem 3.4, the new output is a $(2k-1)$-spanner with $O(n^{1+1/k})$ edges.

The next important point is that, whenever an edge e added into the spanner H, the algorithm never tries to remove e from H. So e remains in H until it is deleted by the adversary. Therefore, the total recourse is $O(m)$. With this, we conclude the following key lemma:

Lemma 3.5. *Given a graph G with n vertices and m initial edges undergoing only edge deletions, the algorithm above maintains a $(2k-1)$-spanner H of G of size $O(n^{1+1/k})$ with $O(m)$ total recourse.*

By plugging Lemma 3.5 to the fully-dynamic-to-decremental reduction by [BKS12] below, we conclude Theorem 3.1. We also include the proof of Lemma 3.6 in Appendix B.2 for completeness.

Lemma 3.6 ([BKS12]). *Suppose that for a graph G with n vertices and m initial edges undergoing only edge deletions, there is an algorithm that maintains a $(2k-1)$-spanner H of size $O(S(n))$ with $O(F(m))$ total recourse where $F(m) = \Omega(m)$, then there exists an algorithm that maintains a $(2k-1)$-spanner H' of size $O(S(n) \log n)$ in a fully dynamic* *graph with n vertices using $O(F((U)\log n))$ total recourse. Here U is the total number of updates, starting from an empty graph.*

3.2. 3-Spanner with Near-optimal Recourse and Fast Update Time

In this section, we prove Theorem 3.2 by showing an algorithm for maintaining a 3-spanner with small update time *in addition* to having small recourse. We start by explaining a basic static construction and needed data structures in Section 3.2.1 and show the dynamic algorithm in Section 3.2.2. Assuming our key lemma (Lemma 3.8) about proactive resampling, most details here are quite straight forward. Hence, some proofs are deferred to Appendix B.3.

Throughout this section, we let $N_G(u) = \{v \in V : (u, v) \in E\}$ denote the set of neighbors of a node $u \in V$ in a graph $G = (V, E)$, and we let $\deg_G(u) = |N_G(u)|$ denote the degree of the node u in the graph G.

3.2.1. A Static Construction and Basic Data Structures

A Static Construction. We now describe our static algorithm. Though our presentation is different, our algorithm is almost identical to [GP17]. The only difference is that we do not distinguish small-degree vertices from large-degree vertices.

We first arbitrarily partition V into $\sqrt{n}$ equal-sized *buckets* $V_1, \ldots, V_{\sqrt{n}}$. We then construct three sets of edges E_1, E_2, E_3. For every bucket $V_i, i \in [1, \sqrt{n}]$, we do the following. First, for all $v \in V \setminus V_i$, if $V_i \cap N_G(v)$ is not empty, we choose a neighbor $c_i(v) \in V_i \cap N_G(v)$ and add $(v, c_i(v))$ to E_1. We call $c_i(v)$ an *i-partner* of v. Next, for every edge $e = (u, v)$, where both $u, v \in V_i$, we add e to E_2. Lastly, for $u, u' \in V_i$ with overlapping neighborhoods, we pick an arbitrary common neighbor $w_{uu'} \in N_G(u) \cap N_G(u')$ and add $(u, w_{uu'}), (w_{uu'}, u')$ to E_3. We refer to the node $w_{uu'}$ as the *witness* for the pair u, u'.

Claim 3.7. *The subgraph $H = (V, E_1 \cup E_2 \cup E_3)$ is a 3-spanner of G consisting of at most $O(n\sqrt{n})$ edges.*

Dynamizing the Construction. Notice that it suffices to separately maintain E_1, E_2, E_3, in order to maintain the above dynamic 3-spanner. Maintaining E_1 and E_2 is straightforward and can be done in a fully-dynamic setting in $O(1)$ worst-case update time. Indeed, if $e = (u, c_i(u)) \in E_1$ is deleted, then we pick a new i-partner $c_i(u) \in V_i \cap N_G(u)$ for u. Maintaining $V_i \cap N_G(u)$ for all u allows us to update $c_i(u)$ efficiently. If $e = (u, u') \in E_2$, where $u, u' \in V_i$, is deleted, then we do nothing.

The remaining task, maintaining E_3, is the most challenging part of our dynamic algorithm. Before we proceed, let us define a subroutine and a data structure needed to implement our algorithm.

Resampling Subroutine. We define Resample(u, u') as a subroutine that *uniformly samples* a witness $w_{uu'}$ (i.e. a common neighbor of u and u'), if exists. Notice that, we can obtain E_3 by calling Resample(u, u') for all $u, u' \in V_i$ and for all $i \in [1, \sqrt{n}]$.

Partnership Data Structures. The subroutine above hints that we need a data structure for maintaining the common neighborhoods for all pairs of vertices that are in the same bucket. For vertices u and v within the same bucket, we let $P(u, v) = N_G(u) \cap N_G(v)$ be the *partnership* between u and v. To maintain these structures dynamically, when an edge (u, v) is inserted, if $u \in V_i$ and $v \in V_j$, we add v to $P(u, u')$ for all $u' \in V_i \cap N_G(v) \setminus \{u\}$, and symmetrically add u to $P(v, v')$ for all $v' \in V_j \cap N_G(u) \setminus \{v\}$. This clearly takes $O(\sqrt{n} \log n)$ worst-case time for edge insertion. and this is symmetric for edge deletion.

As we want to prove that our final update time is $\tilde{O}(\sqrt{n})$, we can assume from now that E_1, E_2, and all partnerships are maintained in the background.

3.2.2. Maintaining Witnesses via Proactive Resampling

Remark. For clarity of exposition, we will present an amortized update time analysis. Using standard approach, we can make the update time worst case. We will discuss this issue at the end of this section.

Our dynamic algorithm runs in *phases*, where each phase lasts for $n\sqrt{n}$ consecutive updates (edge insertions/deletions). As a spanner is decomposable,[3] it suffices to maintain a 3-spanner H of the graph undergoing only edge deletions within this phase and then include all edges inserted within this phase into H, which increases the size of H by at most $n\sqrt{n}$ edges. Henceforth, *we only need to present how to initialize a phase and how to handle edge deletions within each phase.* The reason behind this reduction is because our proactive resampling technique naturally works for decremental graphs.

Initialization. At the start of the phase, since our partnerships structures only processed edge deletions from the previous phase, we first update partnerships with all the $O(n\sqrt{n})$ inserted edges from the previous phase. Then, we call $\textsc{Resample}(u, u')$ for all $u, u' \in V_i$ for all $i \in [1, \sqrt{n}]$ to replace all witnesses and initialize E_3 of this phase.

Difficulty of Bounding Recourse. Maintaining E_3 (equivalently, the witnesses) in $\tilde{O}(\sqrt{n})$ worst-case time is straightforward because the partnership data structure has $O(\sqrt{n}\log n)$ update time. However, our goal is to show $\tilde{O}(1)$ amortized recourse, which is the most challenging part of our analysis. To see the difficulty, if (u, v) is deleted and $u \in V_i$, a vertex v may serve as a witness $\{w_{uu'}\}$ for all $u' \in V_i$. In this case, deleting (u, v) causes the algorithm to find a new witness $w_{uu'}$ for all $u' \in V_i$. This implies a recourse of $|V_i| = \Omega(\sqrt{n})$. To circumvent this issue, we apply the technique of *proactive resampling*, as described below.

[3]Let $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$. If H_1 and H_2 are α-spanners G_1 and G_2 respectively, then $H_1 \cup H_2$ is a α-spanner of $G_1 \cup G_2$.

Proactive Resampling. We keep track of a *time-variable* T; the number of updates to G that have occurred in this phase until now. T is initially 0. We increment T each time an edge gets deleted from G.

In addition, for all $i \in [1, \sqrt{n}]$ and $u, u' \in V_i$ with $u \neq u'$, we maintain: (1) $w_{uu'}$, the *witness* for the pair u and u' and (2) a set $\textsc{Schedule}[u, u']$ of positive integers, which is the set of timesteps where our algorithm intends to *proactively resample* a new witness for u, u'. This set grows adaptively each time the adversary deletes $(u, w_{uu'})$ or $(w_{uu'}, u')$.

Finally, to ensure that the update time of our algorithm remains small, for each $\lambda \in [1, n\sqrt{n}]$ we maintain a $\textsc{List}[\lambda]$, which consists of all those pairs of nodes (x, x') such that $\lambda \in \textsc{Schedule}[x, x']$.

When an edge (u, v), where $u \in V_i$ and $v \in V_j$ is deleted, we do the following operations. First, for all $u' \in V_i$ that had $v = w_{uu'}$ as a common neighbor with u before deleting (u, v), we add the timesteps $\{T + 2^k \mid T + 2^k \leq n\sqrt{n}, k \in \mathbb{N}\}$ to $\textsc{Schedule}[u, u']$. Second, analogous to the previous one, for all $v' \in V_j$ that had $u = w_{vv'}$ as a common neighbor with v before deleting (u, v), we add the timesteps $\{T + 2^k \mid T + 2^k \leq n\sqrt{n}, k \in \mathbb{N}\}$ to $\textsc{Schedule}[v, v']$. Third, we set $T \leftarrow T + 1$. Lastly, for each $(x, x') \in \textsc{List}[T]$, we call the subroutine $\textsc{Resample}(x, x')$.

The key lemma below summarizes a crucial property of this dynamic algorithm. Its proof appears in Section 3.3.

Lemma 3.8. *During a phase consisting of $n\sqrt{n}$ edge deletions, our dynamic algorithm makes at most $\tilde{O}(\sqrt{n})$ calls to the* Resample *subroutine after each edge deletion. Moreover, the* total *number of calls to the* Resample *subroutine during an* entire phase *is at most $\tilde{O}(n\sqrt{n})$ w.h.p. Both these guarantees hold against an adaptive adversary.*

Analysis of Recourse and Update Time. Our analysis are given in the lemmas below.

Lemma 3.9 (Recourse)**.** *The amortized recourse of our algorithm is $\tilde{O}(1)$ w.h.p., against an adaptive adversary.*

Proof. To maintain the edge-sets E_1 and E_2, we pay a worst-case recourse of $O(1)$ per update. For maintaining the edge-set E_3, our total recourse during the entire phase is at most $O(1)$ times the number of calls made to the RESAMPLE(., .) subroutine, which in turn is at most $\tilde{O}(n\sqrt{n})$ w.h.p. (see Lemma 3.8). Finally, while computing E_3 in the beginning of a phase, we pay $O(n\sqrt{n})$ recourse. Therefore, the overall total recourse during an entire phase is $\tilde{O}(n\sqrt{n})$ w.h.p.. Since a phase lasts for $n\sqrt{n}$ time steps, we conclude the lemma. □

Lemma 3.10 (Worst-case Update Time within a Phase)**.** *Within a phase, our algorithm handles a given update in $\tilde{O}(\sqrt{n})$ worst case time w.h.p..*

Proof. Recall that the sets E_1, E_2 can be maintained in $O(1)$ worst case update time. Henceforth, we focus on the time required to maintain the edge-set E_3 after a given update in G.

Excluding the time spent on maintaining the partnership data structure (which is $\tilde{O}(\sqrt{n})$ in the worst-case anyway), this is proportional to $\tilde{O}(1)$ times the number of calls made to the RESAMPLE(., .) subroutine, *plus* $\tilde{O}(1)$ times the number of pairs $u, u'(v, v')$ where we need to adjust SCHEDULE$[u, u']$. The former is w.h.p. at most $\tilde{O}(\sqrt{n})$ according to Lemma 3.8, while the latter is also at most $\tilde{O}(\sqrt{n})$ since $|V_i|, |V_j| \leq \sqrt{n}$. Thus, within a phase we can also maintain the set E_3 w.h.p. in $\tilde{O}(\sqrt{n})$ worst case update time. □

Although the above lemma says that we can handle each edge deletion in $\tilde{O}(\sqrt{n})$ worst-case update time, our current algorithm does not guarantee worst-case update time yet because the intialization time exceed the $\tilde{O}(\sqrt{n})$ bound. In more details, observe that the total initialization time is $O(n\sqrt{n}) \times O(\sqrt{n}\log n)$ because we need to insert $O(n\sqrt{n})$ edges into partnership data structures, which has $O(\sqrt{n}\log n)$ update time. Over a phase of $n\sqrt{n}$ steps, this implies only $\tilde{O}(\sqrt{n})$ amortized update time.

However, since the algorithm takes long time only at the initialization of the phase, but takes $\tilde{O}(\sqrt{n})$ worst-case step for each update during the phase, we can apply the standard building-in-the-background technique (see Appendix B.3.1) to de-amortized the update time. We conclude the following:

Lemma 3.11 (Worst-case Update Time for the Whole Update Sequence). *W.h.p., the worst-case update time of our dynamic algorithm is $\tilde{O}(\sqrt{n})$.*

3.3. Proactive Resampling: Abstraction

The goal of this section is to prove Lemma 3.8 for bounding the recourse of our 3-spanner algorithm. This is the most technical part of this paper. To ease the analysis, we will abstract the problem situation in Lemma 3.8 as a particular dynamic problem of assigning jobs to machines while an adversary keeps deleting machines and the goal is to minimize the total number of reassignments. Below, we formalize this problem and show how to use it to bound the recourse of our 3-spanner algorithm.

Our abstraction has two technical contributions: (1) it allows us to easily work with multiple sampling probabilities, while in [BBG+20], they fixed a single parameter on sampling probability, (2) the simplicity of this abstraction can expose the generality of the proactive resampling technique itself; it is not specific to the cut sparsifier problem as used in [BBG+20].

Jobs, Machines, Routines, Assignments, and Loads. Let J denote a set of *jobs* and M denote a set of *machines*. We think of them as two sets of vertices of the (hyper)-graph $G = (J, M, R)$.[4] A *routine* $r \in R$ is a hyperedge of G such that r contains exactly one job-vertex from J, denoted by $\text{job}(r) \in J$, and may contain several machine-vertices from M, denoted by $M(r) \subseteq M$. Each routine r in G means there is a routine for handling $\text{job}(r)$ by *simultaneously* calling machines in $M(r)$. Note that $r = \{\text{job}(r)\} \cup M(r)$. We

[4]This graph is different from the graph that we maintain a spanner in previous sections.

say that r is a *routine for* job(r). For each machine $x \in M(r)$, we say that routine r *involves* machine x, or that r *contains* x. The set $R(x)$ is then defined as the set of routines involving machine x. Observe that there are $\deg_G(u)$ different routines for handling job u. An *assignment* $A = (J, M, F \subseteq R)$ is simply a subgraph of G. We say assignment A is *feasible* iff $\deg_A(u) = 1$ for every job $u \in J$ where $\deg_G(u) > 0$. That is, every job is handled by some routine, if exists. When $r \in A$, we say that job(r) is *handled by* routine r. Finally, given an assignment A, the *load* of a machine x is the number of routines in A involving x, or in other words, is the degree of x in A, $\deg_A(x)$. We note explicitly that our end-goal is not to optimize loads of machines. Rather, we want to minimize the number of reassignments needed to maintain feasible assignments throughout the process.

In this section, we usually use u, v, w to denote jobs, use x, y, z to denote machines, and use e or r to denote routines or (hyper)edges.

The Dynamic Problem. Our problem is to maintain a feasible assignment A while the graph G undergoes a sequence of machine deletions (which might stop before all machines are deleted). More specifically, when a machine x is deleted, all routines r containing x are deleted as well. But when routines in A are deleted, A might not be feasible anymore and we need to add new edges to A to make A feasible. Our goal is to minimize the total number of routines ever added to A.

To be more precise, write the graph G and the assignment A after t machine-deletions as $G^t = (J, M, R^t)$ and $A^t = (J, M, F^t)$, respectively. Here, we define *recourse* at timestep t to be $|F^t \setminus F^{t-1}|$, which is the number of routined added into A at timestep t. When the adversary deletes T machines, the goal is then to minimize the total recourse $\sum_{t=1}^{T} |F^t \setminus F^{t-1}|$.

The Algorithm: Proactive Resampling. To describe our algorithm, first let $\textsc{Resample}(u)$ denote the process of reassigning job u to a uniformly random routine for u. In the graph language, $\textsc{Resample}(u)$ removes the edge r such that job$(r) = u$ from A, sample an edge

r' from $\{r \in R \mid \text{job}(r) = u\}$, and then add r' into A. At timestep 0, we initialize a feasible assignment A^0 by calling Resample(u) for every job $u \in J$, i.e., assign each job u to a random routine for u. Below, we describe how to handling deletions.

Let T be the total number of machine-deletions. For each job u, we maintain Schedule(u) $\subseteq [T]$ containing all time steps that we will invoke Resample(u). That is, at any timestep t, before an adversary takes any action, we call Resample(u) if $t \in$ Schedule(u).

We say that an adversary *touches* u at timestep t if the routine $r \in A^t$ handling u at time t is deleted. When u is touched, we call Resample(u) and, very importantly, we put $t+1, t+2, t+4, \ldots$ where $t + 2^i \leq T$ into Schedule(u). This is the action that we call *proactive resampling* because we do not just resample a routine for u only when u is touched, but do so proactively in the future as well. This completes the description of the algorithm.

Clearly, A remains a feasible assignment throughout because whenever a job u is touched, we immediately call Resample(u). The key lemma below states that the algorithm has low recourse, even if the adversary is adaptive in the sense that each deletion at time t depends on previous assignment before time t.

Lemma 3.12. *Let T be the total number of machine-deletions. The total recourse of the algorithm running against an adaptive adversary is $O\big(|J| \log(\Delta) \log^2 |M| + T \log^2 |M|\big)$ with high probability where Δ is the maximum degree of any job. Moreover, if the load of a machine never exceeds λ, then our algorithm has $O(\lambda \log T)$ worst-case recourse.*

We will prove Lemma 3.12 in Section 3.4. Before proceeding any further, however, we argue why Lemma 3.12 directly bounds the recourse of our 3-spanner algorithm.

Back to 3-spanners: Proof of Lemma 3.8. It is easy to see that maintaining E_3 in our 3-spanner algorithm can be framed exactly as the job-machine load-balancing problem. Suppose the given graph is $G = (V, E)$ where $n = |V|$ and $m = |E|$. We create a job $j_{uu'}$ for each pair of vertices $u, u' \in V_i$ with $u \neq u'$. For each edge $e \in E$, we create a machine x_e.

Hence, $|J| = O(n^{1.5})$ and $|M| = |E| = m$. For each job, as we want to have a witness $w_{uu'}$, this witness is corresponding to two edges $e = (u, w_{uu'})$ and $e' = (u', w_{uu'})$. Hence, we create a routine $r = (j_{uu'}, e, e')$ for each $u, u' \in V_i$ and a common neighbor $w_{uu'}$. Since there are at most n common neighbors between each u and u', $\Delta = O(n)$. A feasible assignment is then corresponding to finding a witness for each job. Our algorithm that maintains the spanner is also exactly this load-balancing algorithm. Hence, the recourse of the 3-spanner construction follows from Lemma 3.12 where we delete exactly $T = O(n^{1.5})$ machines. As $|J| = O(n^{1.5})$, the total recourse bound then becomes $O(n^{1.5} \log^3 n)$. As $T = O(n^{1.5})$, averaging this bound over all timesteps yields $O(\log^3 n)$ amortized recourse.

3.4. Proactive Resampling: Analysis (Proof of Lemma 3.12)

The first step to prove Lemma 3.12 is to bound the loads of machines x. This is because whenever machine x is deleted, its load of $\deg_A(x)$ would contribute to the total recourse.

What would be the expected load of each machine? For intuition, suppose that the adversary was *oblivious*. Recall that $R(x)$ denote the set of all routines involving machine x. Then, the expected load of machine x would be $\sum_{r \in R(x)} 1/\deg_G(\text{job}(r))$ because each job samples its routine uniformly at random, and this is concentrated with high probability using Chernoff's bound. Although in reality our adversary is *adaptive*, our plan is to still prove that the loads of machines do not exceed its expectation in the oblivious setting too much. This motivates the following definitions.

Definition 3.13. *The* target load *of machine x is* $\text{target}(x) = \sum_{r \in R(x)} 1/\deg_G(\text{job}(r))$. *The target load of x at time t is* $\text{target}^t(x) = \sum_{r \in R^t(x)} 1/\deg_{G^t}(\text{job}(r))$. *An assignment A has* overhead (α, β) *iff* $\deg_A(x) \le \alpha \cdot \text{target}(x) + \beta$ *for every machine $x \in M$.*

Our key technical lemma is to show that, via proactive resampling, the loads of machines are indeed close to its expectation in the oblivious setting. That is, the maintained assignment has small overhead. Recall that T is the total number of machine-deletions.

Lemma 3.14. *With high probability, the assignment A maintained by our algorithm always has overhead $(O(\log T), O(\log |M|))$ even when the adversary is adaptive.*

Before proving Lemma 3.14, we formally show how to use Lemma 3.14 to bound of machine loads implies the total recourse, which proves Lemma 3.12.

Proof of Lemma 3.12. Let T be the total number of deletions. Observe that the total recourse up to time T is precisely the total number of Resample(.) calls up to time T, which in turn is at most the total number of Resample(.) calls put into Schedule(.) since time 1 until time T. Therefore, our strategy is to bound, for each time t, the number of Resample(.) calls *newly generated* at time t. Let x^t be the machine deleted at time t. Observe this is at most $O(\log T) \times \deg_{A^t}(x^t)$ where $\deg_{A^t}(x^t)$ is x^t's load at time t and the $O(\log T)$ factor is because of proactive sampling.

By Lemma 3.14, we have $\deg_{A^t}(x^t) \leq O\left(\log(T)\text{target}^t(x^t) + \log|M|\right)$. Also, we claim that $\sum_{t=1}^{T} \text{target}^t(x^t) = O(|J| \log \Delta)$ where Δ is the maximum degree of jobs (to be proven below). Therefore, the total recourse up to time T is at most

$$O(\log T) \sum_{t=1}^{T} \deg_{A^t}(x^t) \leq O(\log T) \sum_{t=1}^{T} O\left(\log(T)\text{target}^t(x^t) + \log|M|\right)$$
$$\leq O\left(|J| \log(\Delta) \log^2|M| + T\log^2|M|\right)$$

as $T \leq |M|$.

It remains to show that $\sum_{t=1}^{T} \text{target}^t(x^t) = O(|J| \log \Delta)$. Recall that $\text{target}^t(x) = \sum_{r \ni x} \frac{1}{\deg_{G^t}(\text{job}(r))}$. Imagine when machine x^t is deleted at time t. We will show how to charge $\text{target}^t(x^t)$ to jobs with edges connecting to x^t. For each job u with c (hyper)edges connecting to x^t, u's contribution of $\text{target}^t(x^t)$ is $c/\deg_{G^t}(u)$. So we distribute the charge of $c/\deg_{G^t}(u) \leq \frac{1}{\deg_{G^t}(u)} + \frac{1}{\deg_{G^t}(u)-1} + \ldots + \frac{1}{\deg_{G^t}(u)-c+1}$ to u. Since these edges are charged from machine x^t to job u only once, the total charge of each job u at most $\frac{1}{\deg_G(u)} + \frac{1}{\deg_G(u)-1} +$

$\cdots + 1/2 + 1 = O(\log \Delta)$. Since there are $|J|$ jobs, the bound $\sum_{t=1}^{T} \text{target}^t(x^t) = O(|J| \log \Delta)$ follows.

To see that we have worst-case recourse, one can look at any timestep t. There are $O(\log t)$ timesteps that can cause RESAMPLE to be invoked at timestep t, namely, $t-1, t-2, t-4, \ldots$. At each of these timesteps t', one machine is deleted, so the number of RESAMPLE calls added from timestep t' is also bounded by the load of the deleted machine $x_{t'}$, which does not exceed λ. Summing this up, the number of calls we make at timestep t is at most $O(\lambda \log t) = O(\lambda \log T)$. This concludes our proof. $\square$

3.5. Bounding Load (Proof of Lemma 3.14)

Here, we show that the load $\deg_{A^t}(x)$ of every machine x at each time t is small. Some basic notions are needed in the analysis.

Experiments and Relevant Experiments. An *experiment* X is a binary random variable associated with an edge/routine e and time step t, where $X = 1$ iff RESAMPLE(job(e)) is called at time t and e is chosen to handle job(e), among all edges incident to job(e). Observe that $\mathbb{P}[X = 1] = 1/\deg_{G^t}(\text{job}(e))$. Note that each call to RESAMPLE(u) at time t creates new $\deg_{G^t}(u)$ experiments. We order all experiments $X_1, X_2, X_3, \ldots$ by the time of their creation. For convenience, for each experiment X, we let $e(X)$, $t(X)$, and job(X) denote its edge, time of creation, and job respectively.

Next, we define the most important notion in the whole analysis.

Definition 3.15. *For any time t and edge $e \in R^t$ at time t, an experiment X is* (t, e)-relevant *if*

- *$e(X) = e$, and*
- *there is no $t(X) < t' < t$ such that $t' \in$ SCHEDULE$^{t(X)}$(job(e)).*

Moreover, X is (t, x)-relevant if it is (t, e)-relevant and edge $e \in R^t(x)$ is incident to x.

Intuitively, X is a (t, e)-relevant experiment if X could cause e to appear in the assignment A^t at time t. To see why, clearly if $e(X) \neq e$, then X cannot cause e to appear. Otherwise, if $e(X) = e$ but there is $t' \in (t(X), t)$ where $t' \in \textsc{Schedule}^{t(X)}(\text{job}(e))$, then X cannot cause e to appear at time t either. This is because even if X is successful and so e appears at time $t(X)$, then later at time $t' > t(X)$, e will be resampled again, and so X has nothing to do whether e appears at time $t > t'$. With the same intuition, X is (t, x)-relevant if X could contribute to the load $\deg_{A^t}(x)$ of machine x at time t.

It is important to note that, we decide whether X is a (t, e)-relevant based on $\textsc{Schedule}^{t(X)}(\text{job}(e))$ at time $t(X)$. If it was based on $\textsc{Schedule}^{t}(\text{job}(e))$ at time t, then there would be only a single experiment X that is (t, e)-relevant (which is the one with $e(X) = e$ and maximum $t(X) < t$).

According to Definition 3.15, there could be more than one experiments that are (t, e)-relevant. For example, suppose X is (t, e)-relevant. At time $t(X) + 1$, the adversary could touch job(e), hence, adding $t(X) + 2, t(X) + 4, \ldots$ into $\textsc{Schedule}(\text{job}(e))$. Because of this action, there is another experiment X' that is (t, e)-relevant and $t(X') > t(X)$. This motivates the following definition.

Definition 3.16. *Let $Rel(t, e)$ be the random variable denoting the number of (t, e)-relevant experiments, and let $Rel(t, x) = \sum_{e \in R^t(x)} Rel(t, e)$ denote the total number of (t, x)-relevant experiments.*

To simplify the notations in the proof of Lemma 3.14 below, we assume the following.

Assumption 3.17 (The Machine-disjoint Assumption)**.** *For any routines e, e' with* job$(e) =$ job(e')*, then $M(e) \cap M(e') = \emptyset$. That is, the edges adjacent to the same job are machine-disjoint.*

Note that this assumption indeed holds for our 3-spanner application. This is because any two paths of length 2 between a pair of centers u and u' must be edge disjoint in any simple graph. We show how remove this assumption in Appendix B.1, but the notations are more complicated.

Roadmap for Bounding Loads. We are now ready to describe the key steps for bounding the load $\deg_{A^t}(x)$, for any time t and machine x.

First, we write $\mathcal{X}^{(t,x)} = X_1^{(t,x)}, X_2^{(t,x)}, \dots, X_{Rel(t,x)}^{(t,x)}$ as the sequence of all (t,x)-relevant experiments (ordered by time step the experiments are taken). The order in $\mathcal{X}^{(t,x)}$ will be important only later. For now, we write

$$\overline{\deg}_{A^t}(x) = \sum_{X \in \mathcal{X}^{(t,x)}} X,$$

as the total number of success (t,x)-relevant experiments. As any edge e adjacent to x in A^t may appear only because of some successful (t,x)-relevant experiment $X \in \mathcal{X}^{(t,x)}$, we conclude the following:

Lemma 3.18 (Key Step 1)**.** $\deg_{A^t}(x) \leq \overline{\deg}_{A^t}(x)$.

Lemma 3.18 reduces the problem to bounding $\overline{\deg}_{A^t}(x)$. If all (t,x)-relevant experiments $\mathcal{X}^{(t,x)} = \{X_i^{(t,x)}\}_i$ were independent, then we could have easily applied standard concentration bounds to $\overline{\deg}_{A^t}(x) = \sum_{X \in \mathcal{X}^{(t,x)}} X$. Unfortunately, they are not independent as the outcome of earlier experiments can affect the adversary's actions, which in turn affect later experiments.

Our strategy is to relate the sequence $\mathcal{X}^{(t,x)}$ of (t,x)-relevant experiments to another sequence $\hat{\mathcal{X}}^{(t,x)} = \hat{X}_1^{(t,x)}, \hat{X}_2^{(t,x)} \dots, \hat{X}_{Rel(t,x)}^{(t,x)}$ of *independent* random variables defined as follows. For each (t,x)-relevant experiment $\hat{X}_i^{(t,x)}$ where $e = e(\hat{X}_i^{(t,x)})$ and $u = \text{job}(e)$, we carefully define $\hat{X}_i^{(t,x)}$ as an *independent* binary random variable such that

$$\mathbb{P}[\hat{X}_i^{(t,x)} = 1] = 1/\deg_{G^t}(u),$$

which is the probability that RESAMPLE(u) chooses e at time t. We similarly define

$$\widehat{\deg}_{A^t}(x) = \sum_{\hat{X} \in \hat{\mathcal{X}}^{(t,x)}} \hat{X},$$

that sums independent random variables, where each term in the sum is closely related to the corresponding (t, x)-relevant experiments. By our careful choice of $\mathbb{P}[\hat{X}_i^{(t,x)} = 1]$, we can relates $\widehat{\deg}_{A^t}(x)$ to $\overline{\deg}_{A^t}(x)$ via the notion of *stochastic dominance* defined below.

Definition 3.19. *Let Y and Z be two random variables not necessarily defined on the same probability space. We say that Z* stochastically dominates *Y, written as $Y \preceq Z$, if for all $\lambda \in \mathbb{R}$, we have $\mathbb{P}[Y \geq \lambda] \leq \mathbb{P}[Z \geq \lambda]$.*

Our second important step is to prove the following:

Lemma 3.20 (Key Step 2)**.** $\overline{\deg}_{A^t}(x) \preceq \widehat{\deg}_{A^t}(x)$.

Lemma 3.20, which will be proven in Section 3.5.1, reduces the problem to bounding $\widehat{\deg}_{A^t}(x)$, which is indeed relatively easy to bound because it is a sum of independent random variables. The last key step of our proof does exactly this:

Lemma 3.21 (Key Step 3)**.** $\widehat{\deg}_{A^t}(x) \leq 2\log(t)\text{target}^t(x) + O(\log|M|)$ *with probability* $1 - 1/|M|^{10}$.

We prove Lemma 3.21 in Section 3.5.2. Here, we only mention one important point about the proof. The $\log(t)$ factor above follows from the factor the number of (t, e)-relevant experiment is always at most $\mathit{Rel}(t, e) \leq \log(t)$ for any time t and edge e. This property is so crucial and, actually, is what the proactive resampling technique is designed for.

Given three key steps above (Lemmas 3.18, 3.20 and 3.21), we can conclude the proof of Lemma 3.14.

Proof of Lemma 3.14. Recall that we ultimately want to show that, for every timestep t, the maintained assignment A^t has overhead $O(\log(T), \log|M|)$. In other words, for every $t \in T$ and every $x \in M$, we want to show that

$$\deg_{A^t}(x) \leq \text{target}^t(x) \cdot O(\log(T)) + O(\log|M|).$$

By Lemma 3.18, it suffices to show that

$$\overline{\deg}_{A^t}(x) \leq \text{target}^t(x) \cdot O(\log(T)) + O(\log|M|).$$

By Lemmas 3.20 and 3.21,

$$\begin{aligned} &\mathbb{P}[\overline{\deg}_{A^t}(x) \geq 2\log(t)\text{target}^t(x) + O(\log|M|)] \\ &\leq \mathbb{P}[\widehat{\deg}_{A^t}(x) \geq 2\log(t)\text{target}^t(x) + O(\log|M|)] && \text{(Lemma 3.20)} \\ &\leq 1/|M|^{10}. && \text{(Lemma 3.21)} \end{aligned}$$

Now we apply union bound to the probability above. There are $T \leq |M|$ timesteps and $|M|$ machines, hence the probability that a bad event happens is bounded by $\frac{T|M|}{|M|^{10}} = \frac{1}{|M|^8}$. Here, we conclude the proof of Lemma 3.14.

3.5.1. Key Step 2

The goal of this subsection is to prove Lemma 3.20. The following reduces our problem into proving that a certain probabilistic condition holds.

Lemma 3.22 (Lemma 1.8.7(a) [Doe20])**.** *Let $X_1, \ldots, X_n$ be arbitrary boolean random variables and let $X_1^* \ldots X_n^*$ be independent binary random variables. If we have*

$$\mathbb{P}[X_i = 1 | X_1 = x_1, \ldots, X_{i-1} = x_{i-1}] \leq \mathbb{P}[X_i^* = 1]$$

for all $i \in [n]$ *and all* $x_1, \ldots, x_{i-1} \in \{0,1\}$ *with* $\mathbb{P}[X_1 = x_1, \ldots, X_{i-1} = x_{i-1}] > 0$, *then*

$$\sum_{i=1}^{n} X_i \preceq \sum_{i=1}^{n} X_i^*.$$

In light of the above lemma, we will prove that

$$\mathbb{P}[X_i^{(t,x)} = 1 | X_1^{(t,x)}, \ldots, X_{i-1}^{(t,x)}] \le \mathbb{P}[X_i^{(t,x)} = 1] \le \mathbb{P}[\hat{X}_i^{(t,x)} = 1],$$

in Claims 3.24 and 3.23, respectively. This would imply $\sum_{X_i^{(t,x)} \in \mathcal{X}^{(t,x)}} X_i^{(t,x)} \preceq \sum_{\hat{X}_i^{(t,x)} \in \hat{\mathcal{X}}^{(t,x)}} \hat{X}_i^{(t,x)}$ by Lemma 3.22 above, and so $\overline{\deg}_{A^t}(x) \preceq \widehat{\deg}_{A^t}(x)$, completing the proof Lemma 3.20.

Claim 3.23. *For any* (t,x)*-relevant experiment* $X_i^{(t,x)}$, $\mathbb{P}[X_i^{(t,x)} = 1] \le \mathbb{P}[\hat{X}_i^{(t,x)} = 1]$.

Proof. Let $e = e(X_i^{(t,x)})$ and $t' = t(X_i^{(t,x)})$ be the time that the experiment $X_i^{(t,x)}$ is taken. Note that $t' \le t$ as $X_i^{(t,x)}$ is (t,x)-relevant. The claim follows because

$$\mathbb{P}[X_i^{(t,x)} = 1] = \frac{1}{\deg_{G^{t'}}(\text{job}(e))} \le \frac{1}{\deg_{G^t}(\text{job}(e))} = \mathbb{P}[\hat{X}_i^{(t,x)} = 1].$$

The first equality is because $X_i^{(t,x)}$ succeeds iff RESAMPLE(job(e)) chooses e at time t', which happens with probability $1/\deg_{G^{t'}}(\text{job}(e))$. (Note that knowing that $X_i^{(t,x)}$ is (t,x)-relevant does not change the probability that the experiment $X_i^{(t,x)}$ succeeds because we can determine if X_i is (t,x)-relevant depends on information in the past, including $X_1, X_2, \ldots, X_{i-1}$ and the adversary's actions.) The inequality is because $t' \le t$ and G undergoes deletions only. The last equality is by definition of $\hat{X}_i^{(t,x)}$. □

Claim 3.24. $\mathbb{P}[X_i^{(t,x)} = 1 | X_1^{(t,x)}, \ldots, X_{i-1}^{(t,x)}] \le \mathbb{P}[X_i^{(t,x)} = 1]$.

Proof. By Assumption 3.17, this is true simply because knowing the results of the past experiments and other experiments taken at the same timestep as $X_i^{(t,x)}$ cannot increase the probability that $X_i^{(t,x)}$ being 1. Without the assumption, for some i and $j < i$, it is possible that $\mathbb{P}[X_i^{(t,x)} = 1 | X_j^{(t,x)} = 0] > \mathbb{P}[X_i^{(t,x)} = 1]$ if $t(X_i^{(t,x)}) = t(X_j^{(t,x)})$ and $\text{job}(X_i^{(t,x)}) =$

job($X_j^{(t,x)}$), i.e., $X_i^{(t,x)}$ and $X_j^{(t,x)}$ are being sampled with the same Resample(job($X_i^{(t,x)}$) call. $\square$

3.5.2. Key Step 3

In this section, we prove Lemma 3.21. To simplify our proofs, we say that *time t' is (t,e)-relevant* if there is a (t,e)-relevant experiment X created at time $t(X) = t'$. Since, for each time step t', the algorithm can only create one (t,e)-relevant experiment, we have the following observation:

Observation 3.25. *The number of (t,e)-relevant experiments $R(t,e)$ is exactly the number of (t,e)-relevant time steps.*

Now, we state the following crucial lemma. It says that, there are at most $\log(t)$ experiments that are (t,e)-relevant.

Lemma 3.26. *$Rel(t,e) \leq \log(t)$ for every t and $e \in R^t$.*

Proof. By Observation 3.25, we will bound the total number of (t,e)-relevant time steps. Suppose t' is (t,e)-relevant. It suffices to show that if there is another time $t'' > t'$ which is (t,e)-relevant, then $t'' \geq (t'+t)/2$. This means that each consecutive (t,e)-relevant time steps decrease the gap to the fixed time step t by at least half. So this can happen at most $\log(t)$ times.

To prove the claim, observe that $t'' \notin \textsc{Schedule}^{t'}(\text{job}(e))$ as t' is (t,e)-relevant. Hence, the adversary must touch job(e) at some timestep $s \geq t'$. When that happens, we add $s+1, s+2, \ldots$ into $\textsc{Schedule}^{s}(\text{job}(e))$. Let $s' = s + 2^{\log(t-s)-1}$. It is clear that

$$s' \geq s + 2^{\lceil \log(t-s) \rceil - 1} \geq (s+t)/2 \geq (t'+t)/2.$$

Because any timestep in $(t', s']$ cannot be (t, e)-relevant, t'' must be greater than s'. Hence, $t'' \geq (t' + t)/2$ as claimed. $\square$

The above implies that the expected value of $\widehat{\deg}_{A^t}(x)$ is not too far from the target load of x at time t.

Lemma 3.27. $\mathbb{E}[\widehat{\deg}_{A^t}(x)] \leq \log(t) \text{target}^t(x)$.

Proof. We have the following

$$\mathbb{E}[\widehat{\deg}_{A^t}(x)] = \mathbb{E}[\sum_{\hat{X} \in \hat{\mathcal{X}}^{(t,x)}} \hat{X}] \leq \log(t) \sum_{e \in R^t(x)} 1/\deg_{G^t}(\text{job}(e)) = \log(t) \cdot \text{target}^t(x),$$

where the first and last equalities are by definitions of $\widehat{\deg}_{A^t}(x)$ and $\text{target}^t(x)$, respectively. It remains to prove the inequality.

Observe that $\sum_{\hat{X} \in \hat{\mathcal{X}}^{(t,x)} | e(X_i^{(t,x)}) = e} \mathbb{E}[\hat{X}] = \mathit{Rel}(t, e)/\deg_{G^t}(\text{job}(e))$. This is because the number of terms in the sum is exactly the number of (t, e)-relevant experiments $\mathit{Rel}(t, e)$, and we precisely define each $\hat{X}_i^{(t,x)} \in \hat{\mathcal{X}}^{(t,x)}$ where $e = e(X_i^{(t,x)})$ so that $\mathbb{E}[\hat{X}_i^{(t,x)}] = \mathbb{P}[\hat{X}_i^{(t,x)} = 1] = 1/\deg_{G^t}(\text{job}(e))$. Therefore, by Lemma 3.26, we have

$$\mathbb{E}[\sum_{\hat{X} \in \hat{\mathcal{X}}^{(t,x)}} \hat{X}] = \sum_{e \in R^t(x)} \sum_{\hat{X} \in \hat{\mathcal{X}}^{(t,x)} | e(X_i^{(t,x)}) = e} \mathbb{E}[\hat{X}] \leq \log(t) \cdot \sum_{e \in R^t(x)} 1/\deg_{G^t}(\text{job}(e)).$$

$\square$

The last step is to show that the expectation bound from Lemma 3.27 is concentrated. However, since $\text{target}(x)$ for the machine x can be very small ($\ll 1$), it is not enough to use the standard multiplicative Chernoff's bound. Instead, we will apply the version with both additive error and multiplicative error stated below.

Lemma 3.28 (Additive-multiplicative Chernoff's bound [BV14])**.** *Let $X_1, \dots, X_n$ be independent binary random variables. Let $S = \sum_{i=1}^{n} X_i$. Then for all $\delta \in [0,1]$ and $\alpha > 0$,*

$$\mathbb{P}[S \geq (1+\delta)\mathbb{E}[S] + \alpha] \leq \exp(-\frac{\delta\alpha}{3}).$$

Proof of Lemma 3.21. By plugging $\delta = 1$ and $\alpha = 30 \log |M|$ into the above bound, we have

$$\mathbb{P}\left[\widehat{\deg}_{A^t}(x) \geq 2\mathbb{E}[\widehat{\deg}_{A^t}(x)] + 30 \log |M|\right] \leq \exp(-30 \log |M|/3) = |M|^{-10}.$$

This completes the proof of Lemma 3.21 because $\mathbb{E}[\widehat{\deg}_{A^t}(x)] \leq \log(t)\text{target}^t(x)$ by Lemma 3.27.

3.6. Conclusion

In this chapter, we study fully dynamic spanner algorithms against an adaptive adversary. Our algorithm in Theorem 3.1 maintains a spanner with near-optimal stretch-size trade-off using only $O(\log n)$ amortized recourse. This closes the current oblivious-vs-adaptive gap with respect to amortized recourse. Whether the gap can be closed for *worst-case recourse* is an interesting open problem.

The ultimate goal is to show algorithms against an adaptive adversary with polylogarithmic amortized update time or even worst-case. Via the multiplicative weight update framework [Fle00, GK07], such algorithms would imply $O(k)$-approximate multi-commodity flow algorithm with $\tilde{O}(n^{2+1/k})$ time which would in turn improve the state-of-the-art. We made partial progress toward this goal by showing the first dynamic 3-spanner algorithms against an adaptive adversary with $\tilde{O}(\sqrt{n})$ update time in Theorem 3.3 and *simultaneously* with $\tilde{O}(1)$ amortized recourse in Theorem 3.2, improving upon the $O(n)$ amortized update time since the 15-year-old work by [AFI06].

Generalizing our Theorem 3.3 to dynamic $(2k-1)$-spanners of size $\tilde{O}(n^{1+1/k})$, for any $k \geq 2$, is also a very interesting and challenging open question.

3.7. Authors

This chapter was written by Sayan Bhattacharya, Thatchaphol Saranurak, and Pattara Sukprasert. It was published at The Annual European Symposium on Algorithms (ESA) 2022 [BSS22].

CHAPTER 4

Dataset Versioning

Tremendous amount of data is produced daily due to the increasing usage of online collaboration tools for data storage and management: multiple users might collaborate to produce many versions of a raw data set. The management of all these versions, however, has become increasingly challenging in large enterprises. When we have thousands of versions, each of several terabytes, then storing all versions is extremely costly and wasteful. Reducing data storage and data management costs is a major concern for enterprises [MSS$^+$23].

Important not just in online collaborative settings, dataset versioning is also a key concern for enterprise data lakes as well, that are managing huge volumes of customer data [NZM$^+$19]. Often, existing versions of huge tabular datasets might require a few records (or rows) to be modified (for example, product catalogs), thus resulting in a new version for each such modification. This becomes challenging at the terabyte and petabyte scale and storing all of the versions can incur a huge storage and data management cost for the enterprise. Additionally, dataset versioning is a concern for Data Science and Machine learning (and Deep Learning) pipelines as well, as several versions of the data can get generated by the applications of simple transformations on existing data for training and insight generation purposes, thereby increasing the storage and management costs. It is no surprise therefore that data version control is emerging as one of the hot areas in industry [git05, pac16, DVC17, Ter19, Lak20, Dol19], and even popular cloud solution providers like Databricks are now capturing data lineage information that can help in effective data version management [RFT22].

In a pioneering paper, Bhattacherjee et al. [BCH+15] proposed an innovative model capturing the trade-off between *storage* cost and *retrieval* (recreation) cost. The common use case of their model includes version control (e.g., git, mercurial, svn), collaborative data science and analysis, and sharing data sets and intermediate results among data pipelines. The problem studied by the authors can be defined as follows. Given datasets and a subset of the *"deltas"* between them (shown as directed edges connecting versions), find a compact representation that minimizes the overall storage as well as the retrieval costs of the versions. This involves a decision for each version – either we *materialize* it, (store the version explicitly) or we rely on the edit operations to retrieve the version from another materialized version when necessary. The downside of the latter is that to retrieve a version that was not materialized, we will have to incur a computational overhead as well as a delay while the user waits. There are some follow-up works [ZLJG18, DRMK+22, HXL+20]. However, those works either formulate new problems in different use cases [DRMK+22, MSM+22, HXL+20] or implement a system incorporating the feature to store specific versions and deltas [HXL+20, WDL+18, SKS+19]. We will discuss this in more detail in Section 4.1.4.

Fig. 4.1, taken from Bhattacherjee et al. [BCH+15], illustrates the central point through different storage options. (i) shows the input graph, with annotated storage and retrieval costs (for an edge, the retrieval cost indicates the cost to reconstruct the target version given the source version). If the storage size is not a concern, we should store all versions as in (ii). For (iii) and (iv), it is clear that, by storing v_3, we shorten the retrieval times of v_3 and v_5.

This retrieval/storage trade-off leads to the combinatorial problem of minimizing one type of cost, given a constraint on the other. There are variations of our objective function as well: retrieval cost of a solution can be measured by either the maximum or total (or

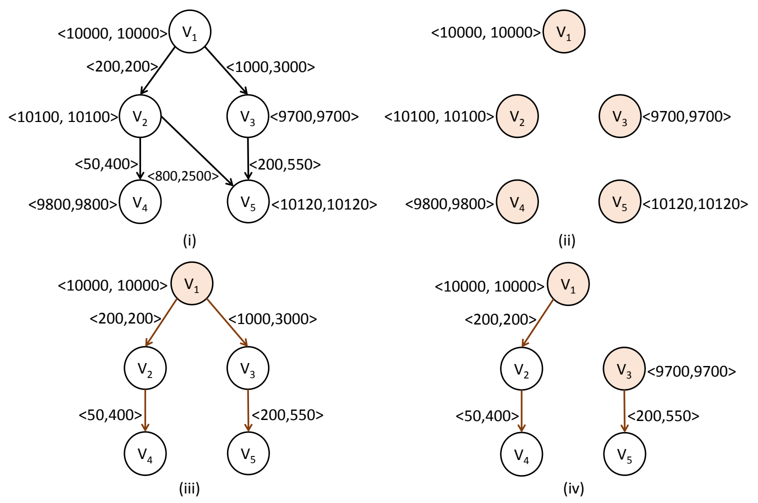


Figure 4.1. (i) A version graph over 5 datasets – annotation ⟨a, b⟩ indicates a storage cost of a and a retrieval cost of b; (ii, iii, iv) three possible storage graphs. The figure is taken from [BCH+15]

equivalently average) retrieval cost of files. This yields four different optimization problems (Problems 3-6 in Table 4.1).

	Problem Name	Storage Cost	Retrieval Cost
Prob. 1	Min Spanning Tree	min	$\mathcal{R}(v) < \infty,\ \forall v$
Prob. 2	Shortest Path Tree	$< \infty$	min $\{\max_v R(v)\}$
Prob. 3	MinSum Retrieval (MSR)	$\leq \mathcal{S}$	min$\{\sum_v R(v)\}$
Prob. 4	MinMax Retrieval (MMR)	$\leq \mathcal{S}$	min $\{\max_v R(v)\}$
Prob. 5	BoundedSum Retrieval (BSR)	min	$\sum_v R(v) \leq \mathcal{R}$
Prob. 6	BoundedMax Retrieval (BMR)	min	$\max_v R(v) \leq \mathcal{R}$

Table 4.1. Problems 1-6

Fundamentally, we can think of the problem as a multi-root arborescence problem in a directed graph. A single-root arborescence in a directed (weighted) graph $G = (V, E)$ is a rooted tree such that every node is reachable from the root, and efficient algorithms for finding a minimum weight arborescence are known.

In our multi-root version, we can imagine that nodes have labels and our goal is to select a subset of nodes (multiple roots) and every node not selected should have a directed path from one of the roots to it. Edges may have multiple costs associated with them, relating to storage and computation (reconstruction) costs. The total sum of the node label values and edge costs directly contributes to the storage cost, and the path length to nodes, directly determines the delay to materialize those versions.

4.0.1. Our Contributions

We provide the first set of provable *inapproximability results* and *approximation algorithms* for the aforementioned optimization problems that trade-off between retrieval and storage costs from different angles.

Hardness. Table 4.2 summarizes all hardness results in this paper. Notably, it's impossible to approximate any of the problems within a constant factor on general directed graphs, with MSR being especially hard. This also motivated the consideration for special graph classes.

Problem	Graph type	Assumptions	Inapproximability
MSR	arborescence	Triangle inequality Single weight[1]	1
	undirected		$1+\frac{1}{e}-\varepsilon$
	general		$\Omega(n)$[2]
MMR	undirected		$2-\varepsilon$
	general		$\log^* n-\omega(1)$
BSR	arborescence		1
	undirected		$(\frac{1}{2}-\varepsilon)\log n$
BMR	undirected		$(1-\varepsilon)\log n$

Table 4.2. Hardness results

Tree Algorithms. We propose algorithms that work well when the given graph is a tree (even this special case is not trivial, as we will see). Specifically, for any graph whose underlying undirected graph is a tree , we show that BMR can be solved exactly and efficiently, and that there exists a $(1+\varepsilon)$-approximation algorithm for MSR.

Inspired by these algorithms on trees, we also proposed new heuristics on general graphs. Compared to the best heuristics in [BCH+15], we improve the MSR solution by several orders of magnitude and the BMR solution by up to 50%.

[1]Both are assumptions in previous work [BCH+15] that simplify the problems. In other words, our hardness results apply even when the weights r and s are equal on the edges, and when the weight satisfies the triangle inequality. These assumptions were not made in any other sections of this paper.
[2]This is true even if we relax $\mathcal{S}$ by $O(\log n)$.

Graphs	Problems	Algorithm	Approx.	Run Time
General Digraph	MSR	LMG-All	heuristic	close to LMG
Bounded Treewidth	MSR & MMR	DP-BTW	$1+\varepsilon$	poly$(n, \frac{1}{\varepsilon})$
	BSR & BMR		$(1, 1+\varepsilon)$	
Bidirectional Tree	MMR	DP-BMR	exact	$n^2 \log \mathcal{R}_{max}$
	BMR			n^2

Table 4.3. Algorithms Summary. Here, $\mathcal{R}_{max}$ is defined to be the maximum retrieval cost between any pair of vertices in the tree.

Bounded Tree-width Algorithms. We extend our approximation algorithms for trees to graphs that are close to trees, as measured by a property called *treewidth* [BB73]. Our extensions to bounded treewidth graphs give $(1+\varepsilon)$-approximation for MSR and MMR, as well as $(1, 1+\varepsilon)$ bi-criteria approximation for BSR and BMR. One motivation behind our attention on bounded treewidth graph is that *Series–parallel graphs*,[3] which are very similar to version graphs derived from repositories, has bounded treewidth. In fact, a graph has treewidth at most two if and only if every biconnected component is a series-parallel graph [Bod98]. To confirm our hypothesis, we measure treewidths from various repositories: datasharing, styleguide, and leetcode have treewidth 2,3, and 6 respectively.

New Heuristic. Additionally for MSR, we show that LMG, the algorithm proposed in [BCH+15], may perform arbitrarily poorly. This behavior is in line with the hardness results and is confirmed by experiments. On the other hand, we propose a slight modification named "LMG-All" (LMGA) which dominates the performance of LMG (by up to around 100 times in certain cases), while exhibiting decent run time on sparse graphs.

Inspired by our algorithms on trees, we also propose two new dynamic programming (DP) heuristics for MSR and BMR respectively. Both algorithms perform extremely well in almost all experiments, even when the input graph is not tree-like.

[3]See, e.g., Eppstein [Epp92] for formal definition.

4.1. Preliminaries

In this section, the definition of the problems, notations, simplifications, and assumptions will be formally introduced.

4.1.1. Problem Setting

In the problems we study, we are given a directed graph $G = (V, E)$. The given graph is a *version graph* where vertices represent *versions* and edges capture *"delta"* between versions. More precisely, every edge $e = (u, v)$ is associated with two weight functions: storage cost s_e and retrieval cost r_e.[4] It takes r_e time to retrieve v given that we have retrieved u. The cost of storing (materializing) e is s_e, and the cost of storing v is s_v. Since there is usually a smallest unit of retrieval/storage cost in real world, we will work with nonnegative integers, that is, $s_e, r_e \in \mathbb{N}$ for all $e \in E$.

In order to retrieve a version v from a materialized version u, there must be some path $P\{(u_{i-1}, u_i)\}_{i=1}^n$ with $u_0 = u, u_n = v$, such that all edges along this path are stored. In such cases, we say that v can be retrieved from u with retrieval cost $\sum_{i=1}^n r_{(u_{i-1},u_i)}$. In the rest of the paper, we say v is "retrieved from u" if u is in the path to retrieve v, and v is "retrieved from materialized u" if in addition u is materialized.

The general optimization goal is to select a set of versions $M \subseteq V$ and a set of edges $F \subseteq E$ of **small** size (w.r.t. storage cost s) such that for each $v \in V \setminus M$, the length of shortest path in F (w.r.t. retrieval cost r) from any node $m \in M$ to v is optimized (different versions of the problem optimize different measures). We denote the cost of this shortest path as $R(v)$.

In some previous works, an **auxiliary root** is added to the graph to simplify the problem. Though not used in our approximation algorithms, this is an important simplification for defining problems and for many previous heuristics. We briefly explain the concept here:

[4]We may use $s_{u,v}$ in place of s_e and $r_{u,v}$ in place of r_e.

instead of finding both set of versions and set of edges, we could simplify the problem by adding an auxiliary root v_{aux} to the graph and let edges in the form (v_{aux}, v) capture the storage cost of storing v explicitly.

More precisely, we generate a graph $G_{aux} = (V_{aux} = V \cup \{v_{aux}\}, E_{aux} = E \cup E')$ from $G = (V, E)$ where $E' = \{(v_{aux}, v) \mid v \in V\}$. Each $e = (v_{aux}, v) \in E_{aux}$ has $s_e = s_v$ and $r_e = 0$. It is straightforward to see that any solution $M \subseteq V, F \subseteq E$ has a 1-to-1 correspondence with a directed spanning tree rooted at v_{aux}. In particular, problems 1 and 2 reduce to familiar problems on G_{aux} (see below).

4.1.2. Problem Definition

Different problems are formulated based on different optimization goals. Recall that, after simplification, we want to select a set of edges $F \subseteq E_{aux}$ such that $H = (V_{aux}, F)$ is an arborescence rooted at v_{aux}. We let $s(H) = \sum_{e \in F} s_e$ be the total storage cost. We also let $R(H) = \sum_{v \in V_{aux}} R(v)$ be the total retrieval cost.

Since the two objectives are negatively correlated, and since we want to capture both aspects, one natural way is to constrain one objective and optimize the other objective. The following optimization goals were originally defined in Bhattacherjee et al. [BCH+15] though we might use different names for brevity. See Table 4.1 for the 6 problem definitions.

Since the first two problems are well studied, we do not discuss them further. In a way, MSR and BSR (MMR and BMR, resp.), are closely related. If we have an algorithm for MSR (MMR, resp.), we can turn it into an algorithm for BSR (BMR, resp.) by binary-searching over $\mathcal{S}$. Vice versa, if we have an algorithm for BSR (BMR, resp.), we can solve MSR (MMR, resp.) by binary-searching over $\mathcal{R}$.

4.1.3. Further Assumptions

Motivated by real world application and tractability, we will introduce some further simplifications or complications. In general, our hardness results (Section 4.2) apply even with the strongest assumptions, namely *undirected graph*, *(generalized) triangle inequality* and *single weight function*. Non-uniform demand is considered a special case among the hardness results. In the algorithm sections (Sections 4.3, 4.4), all our algorithms apply even on *directed graphs* with *two weight functions* that *do not satisfy triangle inequality*. Bob: Changed this. We note that our assumptions are natural and some of them were used in [GN22].

Triangle inequality: It is natural to assume that both weights satisfy triangle inequality, i.e., $r_{u,v} \leq r_{u,w} + r_{w,v}$, since we can always implement the delta $r_{u,v}$ by implementing first $r_{u,w}$ and then $r_{w,v}$. In fact, a more general triangle inequality should hold on G_{aux}, i.e., materializing u and storing (u, v) shouldn't cost less space than materializing v directly.

All hardness results in this paper hold under the generalized triangle inequality.

Directedness: It is possible that for two versions u and v, $r_{u,v} \neq r_{v,u}$. In real world, deletion is also significantly faster and easier to store than addition of content. Therefore, Bhattacherjee et al. [BCH+15] considered both directed and undirected cases; we argue that it is usually more natural to model the problems as directed graphs and focus on that case. Note that in the most general directed setting, it's possible that we are given the delta (u, v) but not (v, u). (or equivalently, $s_{v,u} \geq s_u$)

Single weight function: This is the special case where the storage cost function and retrieval cost function are identical. This can be seen in the real world, for example, when we use simple `diff` to produce deltas. We note all our hardness results hold for single weight functions. All our approximations hold for directed graphs with two weight functions.

Arborescence and trees: An *arborescence*, or a directed spanning tree, is a connected digraph where all vertices except a designated root have in-degree 1, and the root has in-degree 0. If each version is a modification on top of another version, then the "natural" deltas automatically form an arboreal input instance.[5] For practical reasons, we also consider *bi-directional tree* instances, meaning that both (u, v) and (v, u) are available deltas.[6] Empirical evidence shows that having deltas in both direction can greatly improve the quality of the optimal solution.

Bounded treewidth: At a high level, treewidth measures how similar a graph is to a tree [BB73]. As one notable class of graphs with bounded tree-widths, series-parallel graphs highly resemble the version graphs we derive from real-world repositories. Therefore, graphs with bounded treewidth is a natural consideration with high practical utility.

Non-uniform demand Some versions may be requested more often than others. To model this, we may introduce *demands* d_v for $v \in V$, and replace total re-creation cost $\left(\sum_v R(v)\right)$ with *weighted* total re-creation cost $\left(\sum_v d_v R(v)\right)$ in MSR and BSR. This variant, although has great practical value, is not the focus of this paper. We demonstrate a hardness result when demand is non-uniform and hope to address this problem in future works.

4.1.4. Related Works

4.1.4.1. Theory. There has been little theoretical analysis on the exact problems we study. The optimization problems are first formalized in Bhattacherjee et al. [BCH+15], which also compared the effectiveness of several proposed heuristics on both real-world and synthetic data. They defined six variants of the problem, two of which are polynomial-time solvable, and the other four are NP-hard (see Section 4.1.2). Zhang et al.[ZLJG18] followed-up by considering a new objective that's a weighted sum of objectives in MSR and MMR. They also modified the heuristics to fit this objective. There are similar concepts, including *Light Approximate Shortest-path Tree (LAST)* [KRY95] and *Shallow-light Tree (SLT)* [KP97, HKS09, HHZ21, MRS+98, Rav94, KS16]. However, this line of work focuses mainly on undirected graphs and their algorithms don't generalize to the directed

[5]This does not hold true for version controls because of the `merge` operation.
[6]While both edges are available, their storage costs and retrieval costs are not necessarily identical.

case. Among the two problems mentioned, SLT is closely related to MMR and BMR. Here, the goal is to find a tree that is **light** (minimize weight) and **shallow** (bounded depth). To the best of our knowledge, there are only two works that give approximation algorithms for directed shallow-light trees. Chimani and Spoerhase [CS15] gives a bi-criteria $(1+\varepsilon, n^{\varepsilon})$-approximation algorithm that runs in polynomial-time. Their run-time analysis is quite complicated, but it is at least $n^{O(1/\varepsilon)}$. Recently, Ghuge and Nagarajan [GN22] showed that a problem called "submodular tree orienteering" has a $O(\frac{\log n}{\log \log n})$ approximation algorithm that runs in quasi-polynomial time. In this problem, we want to find a directed tree T rooted at r such that $s(T) \leq \mathcal{S}$ and maximize $f(V(T))$ where the objective function f is a submodular function. The authors also extended their algorithm so that it works when both **retrieval costs** and **storage costs** are constrained. Their algorithm can be adapted into $O(\frac{\log^2 n}{\log \log n})$-approximation for MMR and BMR where the approximation part is on the storage cost. For MSR and BMR, their algorithm gives $\left(O(\frac{\log^2 n}{\log \log n}), O(\frac{\log^2 n}{\log \log n})\right)$-approximation. The idea is to run their algorithm for many rounds, where the objective of each round is to **cover as many nodes as possible**. We also note that our assumptions, namely, triangle inequality and integral weights are also used in their paper [GN22].

4.1.4.2. Systems. To implement a system captured by our problems, components spanning multiple lines of works are required. For example, to get a graph structure, one has to keep track of history of changes. This is related to the topic of data provenance [BKT00, SPG$^+$]. Given a graph structure, the question of modeling "deltas" is also of interest. There is a line of work dedicated to studying how to implement `diff` algorithms in different contexts [HVT98, BL98, XJF$^+$14, Mac00, Sue02].

In the case where we have more flexibility, one may think of creating deltas from different versions without much of the change history. However, computing all possible deltas is too wasteful, hence it is necessary to utilize other approaches to identify similar versions/datasets. Such line of work is known in the literature as dataset discovery or dataset similarlity [NZM$^+$19, JJ19, FAK$^+$18, BBN19, BFPK20].

After the work of Bhattacherjee et al. [BCH$^+$15], there are several followup works that implemented systems with a feature that saves only selected versions to reduce redundancy. There are works that focus on version control for relational databases [BW21, HXL$^+$20, SKS$^+$19, WDL$^+$18, CD17, MGE$^+$16, BBC$^+$14, SCMMS12] and works that focus on graph snapshots [YCJ20, KD12, MSM$^+$22]. However, since their focuse was on designing full-fledged systems, the algorithms proposed for these systems are rather simple heuristics, without rigorous theoretical results. Here is a non-exhaustive list of examples. *OrpheusDB* [HXL$^+$20] tackled a similar problem but was designed specifically for relational databases. *Forkbase* [WDL$^+$18] is a version control system for blockchain-like instances with built-in fork semantics. *Pensieve* [YCJ20] is a system designed specifically for storing graphs. Derakhshan et al. [DRMK$^+$22] formulated a more generalized problem, which includes time intervals. However, since they deal with data science & machine learning pipelines, they only consider instances where the underlying graphs are directed acyclic.

4.1.4.3. Usecases. In a version control system such as git, our problem is similar to what `git pack` command aims to do.[7] The original heuristic for `git pack`, as described in an IRC log, is to sort objects in particular order and only create deltas between objects in the same window.[8] It is shown in Bhattacherjee et al. [BCH$^+$15] that git's heuristic does not work well compared to other methods.[9] For svn, the most recent version and deltas to the past versions are stored [Nag06]. Other existing data version management systems include [pac16, DVC17, Ter19, Lak20, Dol19], which offer git-like capabilities suited for

[7]https://www.git-scm.com/docs/git-pack-objects
[8]https://github.com/git/git/blob/master/Documentation/technical/pack-heuristics.txt
[9]There is a blog post that further discusses the point: http://www.cs.umd.edu/~amol/DBGroup/2015/06/26/datahub.html

different use cases, such as data science pipelines in enterprise setting, machine learning-focused, data lake storage, graph visualization, etc.

4.2. Hardness results

We hereby list the main hardness (inapproximability) results of the problems. For completeness, we hereby define the notion of approximation algorithms used in this paper.

Definition 4.1 (ρ-approximation algorithm)**.** *Let $\mathcal{P}$ be a minimization problem where we want to come up with a feasible solution x satisfying some constraints (e.g., $a \cdot x \leq b$). We say that an algorithm $\mathcal{A}$ is a ρ-approximation algorithm for $\mathcal{P}$ if $x_{\mathcal{A}}$, the solution produced by $\mathcal{A}$ is feasible and that $OPT \leq f(x_{\mathcal{A}}) \leq \rho \cdot OPT$ where OPT is an optimal objective value and $f(x)$ is the objective value of a solution x. Here, ρ is an* approximation ratio*. Generally, we want $\mathcal{A}$ to run in polynomial time.*

Definition 4.2 (Polynomial-time approximation scheme (PTAS))**.** ***Polynomial-time approximation scheme (PTAS)*** *A polynomial-time approximation scheme is an algorithm $\mathcal{A}$ that, when given any fixed $\varepsilon > 0$, can produce an $(1 + \varepsilon)$-approximation in time that's polynomial in the instance size. We say that $\mathcal{A}$ is a* fully polynomial-time approximation scheme (FPTAS) *if the runtime of $\mathcal{A}$ is polynomial in both the instance size and $1/\varepsilon$.*

Definition 4.3 (Bi-criteria approximation)**.** *In problems such as ours where optimizing an objective function while meeting all constraints is challenging, we can consider relaxing both aspects. We say that an algorithm $\mathcal{A}$ (α, β)-approximates problem $\mathcal{P}$ if the objective value of its output is at most α times the objective value of an optimal solution* ***and*** *the constraints are violated at most β times.*[10]

[10] We allow $x \leq \beta y$ if the constraint $x \leq y$ is presented.

4.2.1. Heuristics can be Arbitrarily Bad

First, we consider the approximation factor of the best heuristic for MSR in Bhattacherjee et al. [BCH+15], Local Move Greedy (LMG). The gist of this algorithm is to start with the arborescence that minimizes the storage cost, and iteratively materialize a version that most efficiently reduces retrieval cost per unit storage. In other words, in each step, a version is materialized with maximum ρ where $\rho = \frac{\text{reduction in total of retrieval costs}}{\text{increase in storage cost}}$. We provide the pseudo-code for LMG in Algorithm 3.

Algorithm 3: Local Move Greedy (LMG)

```
Input : Extended version graph $G_{aux}$, storage constraint $\mathcal{S}$
$T \leftarrow$ minimum arborescence of $G_{aux}$ rooted at $v_{aux}$ w.r.t. weight function $s$
Let $S(T)$ be the total storage cost of $T$
Let $R(v)$ be the retrieval cost of $v$ in $T$
Let $P(v)$ be the parent of $v$ in $T$
$U \leftarrow V$
while $S(T) < \mathcal{S}$ do
    $(\rho_{max}, v_{max}) \leftarrow (0, \varnothing)$
    for $v \in U$ with $S(T) + s_v - s_{P(v),v} \leq \mathcal{S}$ do
        $T' \leftarrow T \setminus \{(P(v), v)\} \cup \{(v_{aux}, v)\}$
        $\Delta = \sum_v \big(R(v) - R_{T'}(v)\big)$
        if $\Delta/(s_v - s_{P(v),v}) > \rho_{max}$ then
            $\rho_{max} \leftarrow \Delta/(s_v - s_{P(v),v})$
            $v_{max} \leftarrow v$
    $T \leftarrow T \setminus \{(P(v_{max}), v_{max})\} \cup \{(v_{aux}, v_{max})\}$
    $U \leftarrow U \setminus \{v_{max}\}$
    if $U = \varnothing$ then
        return $T$
return $T$
```

Note also that we work with the modified graph G_{aux} with the auxiliary root, as defined in Section 4.1.1. Here we show that, even on simple instances, LMG could perform poorly as an approximation algorithm.

Theorem 4.4. *LMG has an arbitrarily bad approximation factor for* MinSum Retrieval*, even under the following assumptions: (1) G is a directed path; (2) there is a single weight function; and (3) triangle inequality holds.*

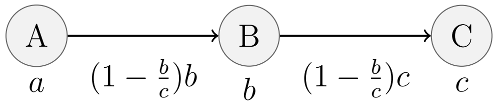


Figure 4.2. An adversarial example for LMG.

Proof. Consider the following chain of three nodes; the storage costs for nodes and the storage/retrieval costs for edges are labeled in Fig. 4.2 (let a be large and $\varepsilon = b/c$ be close to 0). To save space, we do not show v_{aux} but only the nodes of the version graph.

It's easy to check that triangle inequality holds on this graph.

In the first step of LMG, the minimum storage solution of the graph is $\{A, (A, B), (B, C)\}$ with storage cost $a + (1-\varepsilon)b + (1-\varepsilon)c$.

Next, in the greedy step, two options are available: (1). Choosing B and delete (A, B): $\rho_1 = \frac{2(1-\varepsilon)b}{\varepsilon b} = \frac{2}{\varepsilon} - 1$; (2). Choosing C and delete (B, C): $\rho = \frac{(1-\varepsilon)b+(1-\varepsilon)c}{\varepsilon c} = \frac{(1-\varepsilon)b}{b} + \frac{1-\varepsilon}{\varepsilon} = \frac{1}{\varepsilon} - \varepsilon < \frac{2}{\varepsilon} - 1$.

With any storage constraint in range $\big[a + (1-\varepsilon)b + c, a + b + c\big)$, LMG will choose (1) which gives a total retrieval cost of $(1-\varepsilon)c$. Note that with $\mathcal{S} < a + b + c$, LMG is not able to conduct step (2) after taking step (1). However, by choosing (2), which is also feasible, the total retrieval cost is $(1-\varepsilon)b$. The proof is finished by observing c/b can be arbitrarily large. □

4.2.2. Optimizations problems with known hardness

Before we show our hardness results, it is useful to introduce several other NP-hard problems to reduce from.

Definition 4.5 (Set Cover). *Elements $U = \{o_1, \ldots, o_n\}$ and subsets $S_1, \ldots, S_m \subseteq U$ are given. The goal is to find $A \subseteq [m]$ with minimum cardinality such that $\bigcup_{i \in A} S_i = U$.*

Set Cover has no $c \ln n$-approximation for any $c < 1$, unless NP $\subseteq$ DTIME$(n^{O(\log\log n)})$ [Fei98].

Definition 4.6 (Subset Sum)**.** *Given real values $a_1, \ldots, a_n$ and a target value T. The goal is to find $A \subseteq [n]$ such that $\sum_{i \in A} a_i$ is maximized but not greater than T.*

Subset Sum is also NP-hard, but its FPTAS is well studied [IK75, Kar75, GL78, GL94, KMPS03].

Definition 4.7 (k-median and Asymmetric k-median)**.** *Given nodes $V = \{1, \ldots, n\}$, k, and symmetric (resp. asymmetric) distance measures $D_{i,j}$ for $i, j \in V$ that satisfies triangle inequality. The goal is to find a set of nodes $A \subseteq V$ of cardinality at most k that minimizes*

$$\sum_{v \in V} \min_{c \in A} D_{v,c}.$$

The symmetric problem is well studied. The best known approximation lower bound for this problem is $1 + \frac{1}{e}$. We note that an inapproximability result of $1 + \frac{2}{e}$ [JMS02] is often mistakenly quoted for this problem, whereas the authors actually studied the k-median variant where the "facilities" and "clients" are in different sets. With the same method we can only get the hardness of $1 + 1/e$ in our definition.

The asymmetric counterpart is rarely studied. The manuscript [Arc00] showed that there is no (α, β)-approximation (β is the relaxation factor on k) if $\beta \leq \frac{1}{2}(1-\varepsilon)(\ln n - \ln \alpha - O(1))$, unless NP $\subseteq$ DTIME$(n^{O(\log\log n)})$.

Notably, even symmetric $k-$median is inapproximable when triangle inequality is not assumed on the distance measure D. [SO06] However, this hardness is not preserved by the standard reduction to MSR (as in Section 4.2.3.1), since the path distance on graphs inherently satisfies triangle inequality.

Definition 4.8 (k-center and Asymmetric k-center)**.** *Given nodes $V = \{1, \ldots, n\}$, k, and asymmetric distance measures $D_{i,j}$ for $i, j \in V$ that satisfies triangle inequality. The* *goal is to find a set of nodes $A \subseteq V$ of cardinality at most k that minimizes*

$$\max_{v \in V} \min_{c \in A} D_{v,c}.$$

The symmetric problem has a greedy 2-approximation, which is optimal unless P= NP [Gon85].

The asymmetric variant has $\log^* k$ approximation algorithms [Arc01], and one cannot get a better approximation than $\log^* n$ unless NP $\subseteq$ DTIME$(n^{O(\log\log n)})$, if we allow k to be arbitrary [CGH+05].

4.2.3. Hardness Results on General Graphs

In this subsection, we prove the various hardness of approximations on general input graphs. We first focus on MinSum Retrieval and MinMax Retrieval where the constraint is on storage cost and the objective is on the retrieval cost. We then shift our attention to BoundedMax Retrieval and BoundedSum Retrieval in which the constraint is of retrieval cost and the objective function is on minimizing storage cost.

4.2.3.1. Hardness for MinSum Retrieval and MinMax Retrieval.

Theorem 4.9. *On version graphs with n nodes, even assuming single weight function and triangle inequality, there is no:*

(1) *(α, β)-approximation for* MinSum Retrieval *if $\beta \le \frac{1}{2}(1-\varepsilon)\Big(\ln n - \ln \alpha - O(1)\Big)$; in particular, for some constant c, there is no $(c \cdot n)$-approximation without relaxing storage constraint by some $\Omega(\log n)$ factor, unless* NP $\subseteq$ DTIME$(n^{O(\log\log n)})$*;*

(2) *$(1 + \frac{1}{e} - \varepsilon)$-approximation for* MinSum Retrieval *on undirected graphs for all $\varepsilon > 0$, unless* NP $\subseteq$ DTIME$(n^{O(\log\log n)})$*;*

(3) *$\big(\log^*(n) - \omega(1)\big)$-approximation for* MinMax Retrieval*, unless* NP $\subseteq$ DTIME$(n^{O(\log\log n)})$*;*

(4) $(2-\varepsilon)$*-approximation for* MinMax Retrieval *on undirected graphs for all* $\varepsilon > 0$, *unless* NP $=$*P.*

Here, $\log^*(n)$ *denotes the number of logarithms it takes to decrease* n *down to* $3/2$*.*

Proof. MinSum Retrieval. There is an approximation-preserving (AP) reduction[11] from (Asymmetric) k-median to MSR. Let $s_{u,v} = r_{u,v} = d_{u,v}$, the distance from u to v in a (asymmetric) k-median instance. By setting the size of each version v to some large N and storage constraint to be $\mathcal{S} = kN + n$, we can restrict the instance to materialize at most k nodes and retrieve all other nodes through deltas. For large enough N, an (α, β)-approximation for MSR provides an (α, β)-approximation for (Asymmetric) k-median, just by outputting the materialized nodes. The desired results follow from known hardness for asymmetric [Arc00] or symmetric (Section 4.2.2) k-median.

MinMax Retrieval. A similar AP reduction exists from (Asymmetric) k-center to MMR. Again, we can set all materialization costs to N and $c_{u,v} = r_{u,v} = d_{u,v}$, and the desired result follows from the hardness of asymmetric [CGH+05] and symmetric [Gon85] k-center. □

4.2.3.2. Hardness for BoundedSum Retrieval and BoundedMax Retrieval.

Theorem 4.10. *On both directed and undirected version graphs with* n *nodes, even assuming single weight function and triangle inequality, there is no:*

(1) $c_1 \ln n$*-approximation for* BoundedSum Retrieval *for any* $c_1 < 0.5$*;*

(2) $c_2 \ln n$*-approximation for* BoundedMax Retrieval *for any* $c_2 < 1$*.*

unless NP $\subseteq$ DTIME$(n^{O(\log\log n)})$*.*

To prove this theorem, we will present our reduction to these two problems from Set Cover. We then show their structural properties on Lemmas 4.11 and 4.12. We finally show the proof at the end of this section.

[11]See, e.g., [Cre97] for more detail.

Reduction Given a set cover instance with sets $A_1, \dots, A_m$ and elements $o_1, \dots, o_n$, we construct the following version graph:

1. Build versions a_i corresponding to A_i, and b_j corresponding to o_j. All versions have size N for some large $N \in \mathbb{N}$.
2. For all $i, j \in [m], i \neq j$, create symmetric delta (a_i, a_j) of weight 1. For each $o_j \in A_i$, create symmetric delta (a_i, b_j) of weight 1.

Lemma 4.11 (BMR's structure)**.** *Assume we are given an approximate solution to* BMR *on the above version graph under max retrieval constraint $\mathcal{R} = 1$. In polynomial time, we can produce another solution, of equivalent or better quality, such that:*

(1) Only the set versions are materialized. i.e., all $\{b_j\}_{j=1}^n$ are retrieved via deltas.

(2) The storage cost does not exceed that of the original approximate solution, and the maximum retrieval cost is feasible.

Proof of Lemma 4.11. We show (1) by contradiction. Suppose the algorithm produces a solution that materializes b_j.

Case 1: If there exists a_i that needs to be retrieved through b_j (i.e., $o_j \in A_i$), then we can replace the materialization of b_j with that of a_i and replace edges of the form (b_j, a_k) with (a_i, a_k). It is straightforward to see that neither storage cost nor retrieval cost increased in this process.

Case 2: If no other node is dependent on b_j, we can pick any a_i such that (a_i, b_j) exists (again, $o_j \in A_i$). If a_i is already materialized in the original solution, then we can store (a_i, b_j) instead of materializing b_j, which decreases storage cost.

Case 3: If no a_i adjacent to b_j is materialized in the original solution, then some delta $(a_{i'}, a_i)$ has to be stored with the former materialized to satisfy the $\mathcal{R} = 1$ constraint. We can hence materialize a_i, delete the delta $(a_{i'}, a_i)$, and again replace the materialization of b_j with the delta (a_i, b_j) without increasing the storage. Fig. 4.3 illustrates this case. $\square$

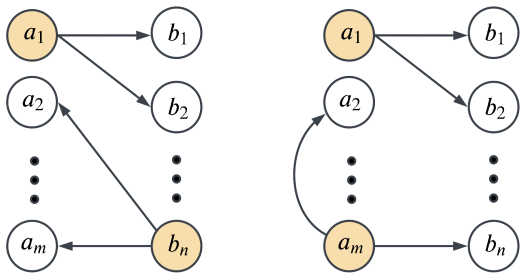


Figure 4.3. Case 1 in proof of Lemma 4.11. The improved solution is on the right.

Lemma 4.12 (BSR's structure)**.** *Assume we are given an approximate solution to* BSR *on the above version graph under total retrieval constraint $\mathcal{R} = m - m_{\text{OPT}} + n$, where m_{OPT} is the size of the optimal set cover. In polynomial time, we can produce an improved solution such that:*

(1) Only the set versions are materialized. i.e., all $\{b_j\}_{j=1}^n$ are retrieved via deltas.

(2) The storage cost does not exceed that of the original approximate solution, and the total retrieval cost is feasible.

Proof of Lemma 4.12. We refer to the same three cases as in Lemma 4.11, and we want to show that, if b_j is materialized,

Case 1: if some a_i is retrieved through b_j, we can apply the same modification as Lemma 4.11. We can replace the materialization of b_j with that of a_i, and replace edges of the form (b_j, a_k) with (a_i, a_k). Neither the storage nor the retrieval cost increases in this case.

Now we WLOG assume no deltas (b_j, a_i) are chosen.

Case 2: if no a_i is retrieved through b_j, and some a_i adjacent to b_j is materialized, then method in Lemma 4.11 needs to be modified a bit in order to remove the materialization of b_j. If we simply retrieve b_j via the delta (a_i, b_j), we would lower the storage cot by $N - 1$ and *increase* the total retrieval cost by 1. This renders the solution infeasible if the total retrieval constraint is already tight.

To tackle this, we analyze the properties of the solutions with total retrieval cost exactly $\mathcal{R}$. Observe that all solutions must materialize at least m_{OPT} nodes at all time, so a configuration exhausting the constraint R must have some version w with retrieval cost at least 2. If this w is a set version, we can loosen the retrieval constraint by storing a delta of cost 1 from some materialized set instead. If w is an element version, then we can materialize its parent version (a set covering it), which increases storage cost by $N-1$ and decreases total retrieval cost by at least 2.

Either case, by performing the above action if necessary, we can resolve case 2 and obtain a approximate solution that's not worse than before.

Case 3: this is where each a_i adjacent to b_j neither retrieves through b_j nor is materialized. Fix an a_i, then some delta $(a_{i'}, a_i)$ has to be stored to retrieve a_i; we can WLOG assume that the former is materialized. We can thus materialize a_i, delete the delta $(a_{i'}, a_i)$, and again replace the materialization of b_j with the delta (a_i, b_j) with no increase in either costs. □

Equipped with Lemma 4.11 and Lemma 4.12, we are now ready to prove Theorem 4.10.

Proof of Theorem 4.10. Assumign $m = O(n)$ in the set cover instance, we present an AP reduction from Set Cover to both BMR and BSR.

BoundedMax Retrieval To produce a set cover solution, we take an improved approximate solution for BMR, and output the family of sets whose corresponding versions are materialized. Since none of the b_j's is stored, they have to be re-created from some a_i. Moreover, under the constraint $\mathcal{R} = 1$, they have to be a 1-hop neighbor of some a_i, meaning the materialized a_i covers all of the elements in the set cover instance.

Finally, we prove that the approximation factor is preserved: for large N, the improved solution has objective value

$$\approx N|\{i : a_i \text{ materialized}\}|.$$

Hence, assuming $n = O(m)$, an $\alpha(|V|)$-approximation for MMR provides a $(\alpha(n) + O(1))$-approximation for set cover. Hence we can't have $\alpha(|V|) = c \ln n$ for $c < 1$ unless NP $\subseteq$ DTIME$(n^{O(\log \log n)})$ [Fei98].

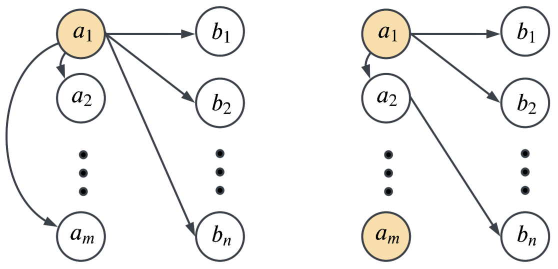


Figure 4.4. The BSR case in proof of Theorem 4.10. The solution on the right has one version (b_2) of retrieval cost 2, hence it must materialize an additional version a_m to satisfy the total retrieval constraint.

BoundedSum Retrieval Assume for the moment that we know m_{OPT}, then we can set total retrieval constraint to be $\mathcal{R} = m - m_{\mathrm{OPT}} + n$, and work with an improved approximate solution. This choice of $\mathcal{R}$ is made so that an optimal solution must materialize the set versions corresponding to a minimum set cover. All other nodes must be retrieved via a single hop.

By Lemma 4.12, we assume all element versions are retrieved from a (not necessarily materialized) set version that covers it. If $m = O(n)$, an $\alpha(|V|)$-approximation of BMR materializes $m_{\mathrm{ALG}} \leq (\alpha(n) + O(1))m_{\mathrm{OPT}}$ nodes.

Note that, by materializing additional nodes, we are allowing a set B of b_j's to have retrieval cost ≥ 2. Let H denote the set of "hopped sets" A_i, which are not materialized yet are necessary to retrieve some b_j through the delta (a_i, b_j). By analyzing the total retrieval cost, we can bound $|H|$ by:

$$|H| \leq |B| \leq m_{\mathrm{ALG}} - m_{\mathrm{OPT}}$$

Specifically, each additional $b_j \in B$ increases retrieval cost by at least 1 compared to the optimal configuration; yet each of the $m_{\mathrm{ALG}} - m_{\mathrm{OPT}}$ additionally materialized set versions only decreases total retrieval cost by 1. It follows that the family of sets

$$S = \{A_i : a_i \text{ materialized } \} \cup H$$

is a $\Big(2\alpha(n) - O(1)\Big)$-approximation solution for the corresponding Set Cover instance. S is feasible because all of the b_j's are retrieved through some (a_i, b_j), where $A_i \in S$; on the other hand, the size of both sets on the right hand side are at most $(\alpha(n) + O(1))m_{\text{OPT}}$, hence the approximation factor holds. Thus, any $\alpha(|V|) = c \ln n$ for any $c < 0.5$ will result in a Set Cover approximation factor of $2c \cdot \ln(n)$.

We finish the proof by noting that, without knowing m_{OPT} in advance, we can run the above procedure for each possible guess of the value m_{OPT}, and obtaining a feasible set cover each iteration. The desired approximation factor is still preserved by outputting the minimum set cover solution over the guesses. □

As a side note, MinSum Retrieval becomes impossibly hard on general graphs when non-uniform demands are allowed:

Theorem 4.13. *On directed version graphs with $r = s$, triangle inequality, and non-uniform demand,* MinSum Retrieval *is inapproximable.*

Proof. This follows from the same reduction from Asymmetric k-median as in Section 4.2.3.1. □

4.2.4. Hardness on Arborescence

We show that MSR and BSR are NP-hard on arborescence instances. This essentially shows that our FPTAS algorithm for MSR in Section 4.4.1 is the best we can do in polynomial time.

Theorem 4.14. *On arborescence inputs,* MinSum Retrieval *and* BoundedSum Retrieval *are* NP*-hard even when we assume single weight function and triangle inequality.*

In order to prove the theorem above, we rely on the following reduction which connects two problems together.

Lemma 4.15. *If there exists poly-time algorithm $\mathcal{A}$ that solves* BoundedSum Retrieval *(resp.* BoundedMax Retrieval*) on some set of input instances, then there exists a poly-time algorithm solving* MinSum Retrieval *(resp.* MinMax Retrieval*) on the same set of input instances.*

Proof. Suppose we want to solve a MSR (resp. MMR) instance with storage constraint $\mathcal{S}$. We can use $\mathcal{A}$ as a subroutine and conduct binary search for the minimum retrieval constraint $\mathcal{R}^*$ under which BSR (resp. BMR) has optimal objective at most $\mathcal{S}$. $\mathcal{R}^*$ is thus an optimal solution for our problem at hand.

To see that the binary search takes poly(n) steps, we note that the search space for the target retrieval constraint is bounded by $n^2 r_{max}$ for BSR and nr_{max} for BMR, where $r_{max} = \max_{e \in E} r_e$. $\square$

Now we show the proof for Theorem 4.14.

Proof of Theorem 4.14. Assuming Lemma 4.15, it suffices to show the NP-hardness of MSR on these inputs.

Consider an instance of Subset Sum problem with values $a_1, \ldots, a_n$ and target T. This problem can be reduced to MSR on an n-nary arborescence of depth one. Let the root version be v_0 and its children $v_1, \ldots, v_n$. The materialization cost of v_i is set to be $a_i + 1$ for $i \in [n]$, while that of v_0 is some N large enough so that the generalized triangle inequality holds. For each $i \in [n]$, we can set both retrieval and storage costs of edge (v_0, v_i) to be 1.

Consider MSR on this graph with storage constraint $\mathcal{S} = N + n + T$. From an optimal solution, we can construct set $A = \{i \in [n] : v_i \text{ materialized}\}$, an optimal solution for the above Subset Sum instance. $\square$

4.3. Exact Algorithm for MMR and BMR on bi-directional trees

By Lemma 4.15, we can use an algorithm for BMR to solve for MMR. Therefore, in this section, it suffices to focus on the problem BMR, namely, we are given constraint $\mathcal{R}$ on the maximal retrieval cost, and we want to minimize the total storage cost. We refer to Algorithm 4 for the pseudo code of the algorithm.

Algorithm 4: DP-BMR

```
   Input : $T$, a bidirectional tree and $\mathcal{R}$, the max retrieval cost constraint
1  Orient $T$ arbitrarily, and sort $V$ in reverse topological order
2  $\mathrm{DP}[v][u] \leftarrow \infty$ for all $v, u \in V$
3  for $v$ in $V$ do
4      for $u$ in $V$ such that $R(u, v) \leq \mathcal{R}$ do
5          if $u = v$ then
6              $\mathrm{DP}[v][u] \leftarrow s_v$
7          else
8              $\mathrm{DP}[v][u] \leftarrow s_{p[v],v}$, where $p[v]$ is the node before $v$ on the path from $u$ to $v$
9          for $w$ child of $v$ do
10             if $w$ in the path from $u$ to $v$ then
11                 $\mathrm{DP}[v][u] \leftarrow DP[v][u] + \mathrm{DP}[w][u]$
12             else
13                 $\mathrm{DP}[v][u] \leftarrow DP[v][u] + \min\{OPT[w], \mathrm{DP}[w][u]\}$
14         $OPT[v] \leftarrow \min\{DP[v][w] : w \in V(T_{[v]})\}$
15 return OPT$[v_{root}]$
```

Let $T = (V, E)$ be a bi-directional tree instance (abbreviated "tree" in the rest of the section) with given maximum retrieval cost constraint $\mathcal{R}$. We can arbitrarily pick a vertex v_{root} as root, and orient the tree such that the root has no parent, while all other nodes have exactly one parent. This process is straightforward, so we will assume that the given tree is rooted for the rest of the section.

For some $v \in V$, let $T_{[v]}$ denote the subtree of T rooted at v. If v is retrieved from materialized u, we use p_v^u to denote the parent of v on the unique $u - v$ path to retrieve v. We write $p_v^v = v$. We now describe a dynamic programming (DP) algorithm DP-BMR that solves BMR exactly on T.

DP variables. For $u, v \in V$, we define $\mathrm{DP}[v][u]$ to be the minimum storage cost of a *partial solution* on $T_{[v]}$ with respect to version u. The partial solution is defined as a solution with all descendants of v are retrieved from some materialized node in $T_{[v]}$, while v is retrieved from a materialized version u, *potentially outside of the subtree* $T_{[v]}$. See Fig. 4.5 for an illustration.

Importantly, also note that when calculating the storage cost for $DP[v][u]$, if u is not a part of $T_{[v]}$, the incident edge (p_v^u, v) is involved in the calculation, while other edges in the $u - v$ path, or the cost to materialize u, are not involved in it.

Base case. We iterate from the leaves up. Let $R(u, v)$ denote the retrieval cost of the path from u to v. For a leaf v, we set $DP[v][v] = s_v$, and $DP[v][u] = s_{(p_v^u, v)}$ for all $u \neq v$ with $R(u, v) \leq \mathcal{R}$. Here, p_v^u is just the unique parent of v in the tree structure.

All choices of u, v such that $R(u, v) > \mathcal{R}$ are infeasible, and we therefore set $\mathrm{DP}[v][u] = \infty$ in these cases.

Recurrence. For convenience, we define a helper variable $OPT[v]$ to be the minimum storage cost on the subproblem $T_{[v]}$, such that v *is either materialized or retrieved from one of its descendants.*[12] In other words,

$$OPT[v] = \min\{DP[v][w] : w \in V(T_{[v]})\}$$

For recurrence on $\mathrm{DP}[v][u]$ such that $R(v, u) \leq \mathcal{R}$, there are three possible cases of the relationship between v and u (see Fig. 4.5 for illustration). In each case, we outline what we add to $\mathrm{DP}[v][u]$ below.

Case 1. If $u = v$, we materialize v, and each child w of v can be either materialized, retrieved from their materialized descendants, or retrieved from the materialized $u = v$ (u for the other two cases). Note that this is exactly $\min\{OPT[w], \mathrm{DP}[w][u]\}$, and similar facts hold for the following two cases as well.

[12]Note that the case where v is retrieved from u outside of $T_{[v]}$ is not considered in this helper variable.

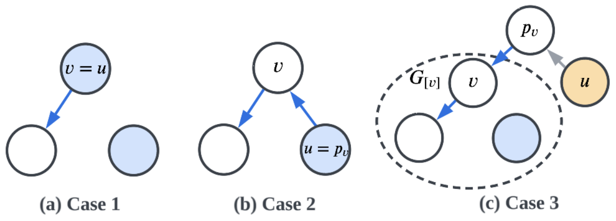


Figure 4.5. 3 cases of DP-BMR. The blue nodes and edges are stored in the partial solution.

Case 2. If $u \in V(T_{[v]})\backslash\{v\}$, we would store the edge (p_v^u, v). Note that p_v^u is a child of v and hence is also retrieved from the materialized u, so we must add $\text{DP}[p_v^u][u]$. We then add $\min\{OPT[w], \text{DP}[w][u]\}$ for all other children w of v.

Case 3. If $u \notin V(T_{[v]})$, we add the edge (p_v^u, v), where p_v^u is the parent of v in the tree structure. We then add $\min\{OPT[w], \text{DP}[w][u]\}$ for all children as before.

Output We output $\text{OPT}[v_{root}]$, which is the storage cost of the optimal solution. To output the configuration achieving this optimum, we can use the standard procedure where we store the configuration at each DP.

Theorem 4.16. BOUNDEDMAX RETRIEVAL *is solvable on bidirectional tree instances in* $O(n^2)$ *time.*

Proof. Time complexity

It is straightforward to see that, for each v, u, computing $DP[v][u]$ takes $O(deg(v))$ time. Hence, computing all the dynamic programming table takes

$$\sum_{u \in V} \sum_{v \in V} deg(v) = \sum_{u \in V} O(n) = O(n^2).$$

Moreover, it takes $O(n^2)$ time to compute the values $R(u, v)$ on a tree. We thus conclude that DP-BMR runs in $O(n^2)$ time.

Optimality We will show by induction that our DP table calculates optimal solution corresponding to each state, i.e., $DP[v][u]$ represents the total storage cost needed for $T_{[v]}$

given that v is retrieved from materialized u. Note that if $u \notin T_{[v]}$, then only the edge (p_v^u, v) is considered in $DP[v][u]$ among all edges in $u - v$ path in T.

In the base case on each leaf v, we set $DP[v][v] = s_v$, and $DP[v][u] = s_{(p_v^u, v)}$ if $u - v$ path has length at most $\mathcal{R}$. This is consistent with the optimal storage cost on the trivial subproblems.

Inductively, suppose we want to compute $DP[v][u]$. Notice that the storage needed for two children w, w' are independent from each other, implying that we can consider them separately. For each child w, if $u \notin T_{[w]}$, then w can either be retrieved through u or some other node in $T_{[w]}$. Hence, we add to $DP[v][u]$ the minimum between $OPT[w] = \min_{u' \in T_{[w]}} DP[w][u']$ and $DP[w][u]$. Otherwise, if $u \in T_{[w]}$, in order for v to be retrieved through u, w has to be retrieved through u, so we add $DP[w][u]$ to $DP[v][u]$. Finally, we increase DP$[v][u]$ by $s_{(p_v^u, v)}$ if $u \neq v$ and s_v if $u = v$. These are all possible cases. Given that $DP[w][u]$ is computed correctly, for all $w \in T_{[v]}$ and $u \in V$, then we compute $DP[v][u]$ correctly.

By induction, we conclude that the DP table is computed correctly. Since the table can capture all feasible solutions, it must capture an optimal solution as well. Hence, our algorithm outputs an optimal answer. $\square$

4.4. Fully polynomial time approximation scheme for MSR via Dynamic Programming

In this section we focus on problem MSR and present a fully polynomial time approximation scheme (FPTAS) on digraphs whose *underlying undirected graph* has bounded treewidth. Similar techniques can be extended to all three other versions of the problems. However, we focus on the method itself and omit the details for the other three problems due to space constraints.

We start by describing a dynamic programming (DP) algorithm on trees in Section 4.4.1. In Section 4.4.2, we define all notations necessary for the latter subsection. Finally, in Section 4.4.3, we show how to extend our DP to the bounded treewidth graphs.

4.4.1. Warm-up: Bidirectional Trees

As shown in Theorem 4.4, the previously best-working LMG algorithm performs arbitrarily badly even on directed paths. In this section, as a warm-up to the more general algorithm in Section 4.4.3, we will present an FPTAS for bidirectional tree instances of MSR via dynamic programming. This algorithm also inspired a practical heuristic DP-MSR, presented in section Section 4.6.2.3.

We can WLOG assume the tree has a designated root v_{root} and a parent-child hierarchy. We can further assume that the tree is binary, via the standard trick of vertex splitting and adding edges of zero weight if necessary. See Appendix C.1.1 for details.

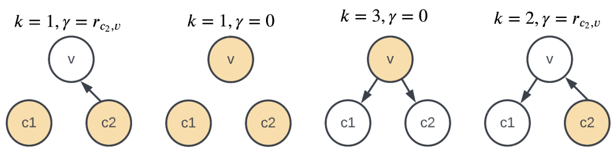


Figure 4.6. An illustration of DP variables in Section 4.4.1

DP variables. We explain the DP variables defined for MinSum Retrieval here: we define DP$[v][k][\gamma][\rho]$ to be the minimum storage cost for the subproblem with constraints v, k, γ, ρ such that (with examples illustrated in Fig. 4.6)

(1) *Root for subproblem* $v \in V$ is a version on the tree; in each iteration, we consider the subtree rooted at v.

(2) *Dependency number* $k \in \mathbb{N}$ stands for the number of versions that will be retrieved via v (including v itself) in the subproblem solution. This is useful when calculating the extra retrieval cost incurred by retrieving v from its parent.

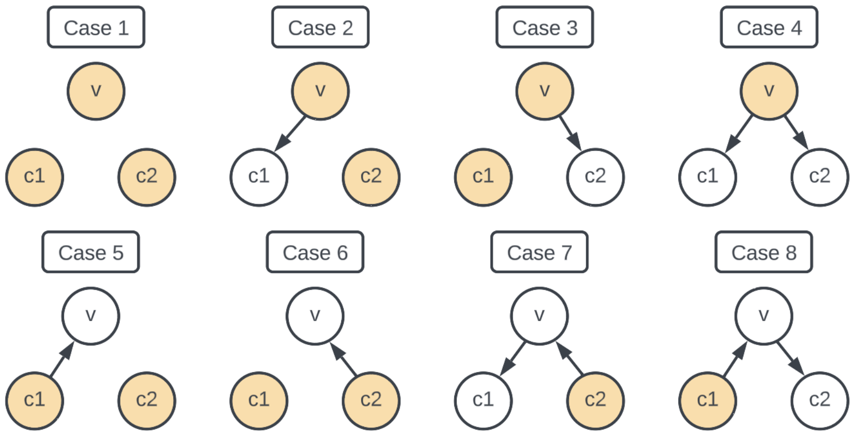


Figure 4.7. Eight types of connections on a binary tree. A node is colored if it is materialized or retrieved via delta from outside the chart. Otherwise, an uncolored node is retrieved from another node as illustrated with the arrows.

(3) *Root retrieval* $\gamma \in \mathbb{N}$ represents the cost of retrieving version v, the root in the subproblem. This is useful when calculating the extra retrieval cost incurred by retrieving the parent of v from v. Note that the root retrieval cost will be discretized, as specified later.

(4) *Total retrieval* $\rho \in \mathbb{N}$ represents the total retrieval cost of the subsolution. Similar to γ, ρ will also be discretized.

Discretizing retrieval costs. Let $r_{max} = \max_{e \in E}\{r_e\}$. The possible total retrieval cost ρ is within range $\{0, 1, \dots, n^2 r_{max}\}$. To make the DP tractable, we partition this range further and define *approximated retrieval cost* $r'_{u,v}$ for edge $(u, v) \in E$ as follows:

$$r'_{u,v} = \lceil \frac{r_{u,v}}{l} \rceil \quad \text{where } l = \frac{n^2 r_{max}}{T(\varepsilon)},\ T(\varepsilon) = \frac{n^4}{\varepsilon},$$

and $T(\varepsilon)$ is the number of "ticks" we want to partition the retrieval range into. The specific choice for $T(\varepsilon)$ will become useful in the proof for Theorem 4.18. We will work with r' in the rest of the subsection.

Base case For a leaf v, we let $DP[v][1][0][0] = s_v$.

Recurrence step For each iteration, we take the minimum over all possible situations as illustrated in Fig. 4.7. The recurrence relation for all cases is given in Appendix C.1.2, and explained in detail for the representative cases below:

4.4.1.1. Dealing with dependency. This refers to the case where a child is to be retrieved from its parent v. Consider case 2 in Fig. 4.7 as an example. Note that $\gamma = 0$ in case 2 since v is materialized. The minimum storage cost in case 2 (given $v, k, \gamma = 0, \rho$) is:

$$
\begin{aligned}
S_2 = {} & s_v + s_{v,c_1} - s_{c_1} \\
& + \min_{\rho_1 \le \rho} \Big\{ DP[c_1][k-1][0][\rho_1 - (k-1)r'_{v,c_1}] && (4.1) \\
& + \min_{k',\gamma_2} \{DP[c_2][k'][\gamma_2][\rho - \rho_1]\} \Big\} && (4.2)
\end{aligned}
$$

In Eq. (4.1), note the dependency number for c_1 needs to be $k-1$ for that of v to be k. The choice of ρ_1 determines how we are allocating retrieval costs budget ρ to c_1 and c_2 respectively. Specifically, the total retrieval cost allocated to subproblem on $G_{[c_1]}$ is $\rho_1 - (k-1) \cdot r'_{v,c_1}$ since an extra $(k-1) \cdot r'_{v,c_1}$ cost is incurred by the edge (v, c_1), as it is used $(k-1)$ times by all versions depending on c_1.

In Eq. (4.2), for a given choice of ρ_1, we want to find the minimum storage cost over the dependency and retrieval cost of c_2. Minimizing this sum of Eqs. (4.1) and (4.2) over all possible combinations of budget splitting yields the minimum storage cost for case 2.

Uprooting We introduce *Uprooting*, a process extensively used in the next section. In the above example, the subproblem solution on $G_{[c_1]}$ materializes c_1, yet in case 2 we would replace this materialization with the diff (v, c_1). This explains the $-s_{c_1}$ term in the equation for S_2.

In general, the restriction of a global solution on a subproblem $G_{[v]}$ does not result in a feasible *partial solution*, due to the possibility of some $v \in V(G_{[z]})$ that's retrieved from versions outside of $G_{[z]}$. The uprooting process allows us to utilize the DP variables on the subproblem in this case. Conversely, by reversing this process (*un-uproot*), we can check if a subproblem is *compatible* with a bigger problem. We explain this in more detail on Section 4.4.3.

Distributing dependency An additional complication arise in case 4, where v is required to have dependency number k and root retrieval 0. For each $k_1 + k_2 = k - 1$, we must go through subproblems where c_1 has dependency number k_1 and c_2 has that of k_2.

4.4.1.2. Dealing with retrieval. In contrast with dependencies, this refers to the case where v is retrieved from one of its children. We take case 5 as an example: given $v, k = 0, \gamma, \rho$,

$$\begin{aligned} S_5 &= s_{c_1,v} \\ &+ \min_{\rho_1 \le \rho} \Bigg\{ \min_{k_1} \{DP[c_1][k_1][\gamma - r_{c_1,v}][\rho_1 - \gamma]\} \\ &+ \min_{k_2,\gamma'} \{DP[c_2][k_2][\gamma'][\rho - \rho_1]\} \Bigg\} \end{aligned}$$

We allocate the retrieval cost similar to case 2. We will care less about the dependency number, over which we will take minimum. The retrieval cost for c_1 now has to be $\gamma - r_{c_1,v}$ since v has to be retrieved from c_1. Note importantly that now we are counting the retrieval cost for v in ρ_1, and so the retrieval cost remaining for the left subproblem now is $\rho_1 - \gamma$. Notice that since only one way of retrieving v will be stored, this retrieval cost will not be over-counted in any cases.

Similarly, we take minimum on all other unused parameters to get the best storage for case 5.

4.4.1.3. Combining the ideas. We take case 8 as an example where both retrieval and dependencies are involved. In case 8, v is retrieved from child c_1 (retrieval), and child c_2 is retrieved from v (dependency). Given v, k, γ, ρ, we claim that:

$$\begin{aligned} S_8 = {} & s_{c_1,v} + s_{v,c_2} - s_{c_2} \\ & + \min_{\rho_1+\rho_2=\rho} \left\{ \min_{k'} \{DP[c_1][k'][\gamma - r_{c_1,v}][\rho_1 - \gamma]\} \right. \\ & \left. + DP[c_2][k-1][0][\rho_2 - (k-1)\cdot(r_2+\gamma)] \right\} \end{aligned}$$

Note that the c_1 side is identical to that for case 5. In combining both dependency and retrieval cases, there is slight adjustment in the dependency side: since v now might also depend on nodes further down c_1 side, the total extra retrieval cost created by adding edge (v, c_2) becomes $(k-1)\cdot(r_2+\gamma)$ instead of $(k-1)\cdot(r_2)$.

Output Finally, with storage constraint $\mathcal{S}$ and root of the tree v_{root}, we output the configuration that outputs the minimum ρ which achieves the following

$$\exists k \le n, \gamma \in \mathbb{N} \quad \text{s.t.} \quad DP[v_{root}][k][\gamma][\rho] \le \mathcal{S}$$

We shall formally state and prove the FPTAS result below.

Lemma 4.17. *The DP algorithm can output a configuration with total retrieval cost at most* $\mathrm{OPT} + \varepsilon r_{max}$ *in poly*$(n, 1/\varepsilon)$ *time.*

Proof. By setting $T(\varepsilon) = \frac{n^4}{\varepsilon}$, we have $l = \frac{n^2 r_{max}}{T(\varepsilon)} = \frac{\varepsilon r_{max}}{n^2}$. Note that we can get an approximation of the original retrieval costs by multiplying each r'_e with l. This creates an estimation error of at most l on each edge. Note further that in the optimal solution, at most n^2 edges are materialized, so if ρ^* is the minimal discretized total retrieval cost, we have

$$\text{total retrieval of DP output} \le l\rho^* \le \text{OPT} + n^2 l \le \text{OPT} + \varepsilon r_{max}$$

□

Now we restate Theorem 4.18:

Theorem 4.18. *For all $\varepsilon > 0$, there is a $(1+\varepsilon)$-approximation algorithm for* MinSum Retrieval *on bidirectional trees that runs in poly$(n, \frac{1}{\varepsilon})$ time.*

Proof. Given parameter ε, we can use the DP algorithm as a black box and iterate the following for up to n times:

(1) Run the DP for the given ε on the current graph. Record the output.

(2) Let (u, v) be the most retrieval cost-heavy edge. We now set $r_{(u,v)} = 0$ and $s_{(u,v)} = s_v$. If the new graph is infeasible for the given storage constraint, or if all edges have already been modified, exit the loop.

At the end, we output the best out of all recorded outputs. This improves the previous bound when $r_{max} > \text{OPT}$: at some point we will eventually have $r_{max} \le \text{OPT}$, which means the output configuration, if mapped back to the original input, is a feasible $(1+\varepsilon)$-approximation. □

4.4.2. Treewidth-Related Definitions

We now consider a more general class of version graphs: any G whose *underlying undirected graph* G_0 has treewidth bounded by some constant k.

Definition 4.19 (Tree Decomposition [BB73])**.** *A tree decomposition of an undirected graph $G_0 = (V_0, E_0)$ is a tree $T = (V_T, E_T)$, where each $z \in V_T$ is associated with a subset ("bag") S_z of V_0. s The bags must satisfy the following conditions:*

(1) $\bigcup_{z \subseteq V_T} S_z = V_0$*;*

(2) For each $v \in V_0$, the bags containing v induce a connected subtree of T;

(3) For each $(u, v) \in E_0$, there exists $z \in V_T$ such that S_z contains both u and v.

The width *of a tree decomposition $T = (V_T, E_T)$ is $\max_{z \in V_T} |S_z| - 1$.*

The treewidth *of undirected graph G_0 is the minimum width over all tree decompositions of G_0.*

It follows that undirected forests have treewidth 1. We further note that there is also a notion of directed treewidth [JRST01], but it's not suitable for our purpose.

For our purpose, we will WLOG assume a special kind of tree decomposition:

Definition 4.20 (Nice Tree Decomposition [Bod98])**.** *A nice tree decomposition is a tree decomposition with a designated root, where each node is one of the following types:*

*(1) a **leaf**, which has no children;*

*(2) a **separator**, which has one child, and whose bag is a subset of the child's bag;*

*(3) a **join**, which has two children, and whose bag is exactly the union of its children's bags.*

Given a bound k on the treewidth, there are multiple algorithms for calculating a desired tree decomposition of width k [Bod93, ACP87, FTV15], or an approximation of k [BF21, FLS+18, Kor22, FHL05]. For our case, the algorithm by Bodlaender [ACP87] can be used to compute a tree decomposition in time $2^{O(k^3)} \cdot O(n)$, which is polynomial in n if the treewidth k of the given input is constant. Given such a tree decomposition, we can in $O(|V_0|)$ time find a nice tree decomposition of the same width with $O(k|V_0|)$ nodes [Bod98].

4.4.3. Generalized Dynamic Programming

Here we outline the DP for MSR on graphs whose underlying undirected graph G_0 has treewidth at most $k - 1$. In order to extend our algorithm in Section 4.4.1 to bounded treewidth graphs, we utilize the techniques from Hajiaghayi [Haj01].

4.4.3.1. DP States. Similar to the warm-up, we will do the DP bottom-up on each $z \in V_T$ in the nice tree decomposition T. When we are at node z, let $V_{[z]} = \bigcup_{z' \in V(T_{[z]})} S_{z'}$ denote the set of vertices that were already considered bags up to bag S_z, including S_z. We now define the *DP states.* At a high level, each state describes some number of *partial solutions* on the subgraph $V_{[z]}$. When building a complete solution on G from the partial solutions, the state variables should give us *all* the information we need.

Each DP state on $z \in V_T$ consists of a tuple of functions

$$\mathcal{T}_z = (\mathrm{Par}_z, \mathrm{Dep}_z, \mathrm{Ret}_z, \mathrm{Anc}_z)$$

and a natural number ρ_z:

(1) *Parent function* $\mathrm{Par}_z : S_z \mapsto V_{[z]}$ describing the partial solution restricted on S_z. If $\mathrm{Par}_z(v) \neq v$ then v will be retrieved through the edge $(\mathrm{Par}_z(v), v)$. If $\mathrm{Par}_z(v) = v$ then v will be materialized.

(2) *Dependency function* $\mathrm{Dep}_z : S_z \mapsto [n]$. Similar to the dependency parameter in the warm-up, $\mathrm{Dep}_z(v)$ counts the number of nodes whose retrieval requires the retrieval of v.

(3) *Retrieval cost function* $\mathrm{Ret}_z : S_z \mapsto \{0, \ldots, K\}$. Similar to the root retrieval parameter in the warm-up, $\mathrm{Ret}_z(v)$ denotes the retrieval cost of version v in the partial solution on $V_{[z]}$.

(4) *Ancestor function* $\mathrm{Anc}_z : S_z \mapsto 2^{S_z}$. $u \in \mathrm{Anc}_z(v)$ denotes that u is retrieved in order to retrieve v. i.e. v is dependent on u. Different from the tree case, to produce a spanning forest here, we need this extra information to avoid directed cycles.

(5) ρ_z, the total retrieval cost of the subproblem according to the partial solution. Similar to its counterpart in the warm-up, ρ_z will be discretized by the same technique that makes the approximation an FPTAS.

A feasible state on $z \in V_T$ is a pair $(\mathcal{T}_z, \rho_z)$ as defined, which correctly describes some partial solution on $V_{[z]}$ whose retrieval cost is exactly ρ_z. Each state is further associated with a storage value $\sigma(\mathcal{T}_z, \rho_z) \in \mathbb{Z}^+$, indicating the minimum storage needed to achieve the state $(\mathcal{T}_z, \rho_z)$ on $V_{[z]}$. We call a minimum-storage solution "the partial solution $\mathcal{T}_z$" for convenience.

Since we have described our states, we are now ready to describe how to compute each state.

4.4.3.2. Recurrence on leaves. For each leaf $z \in V_T$, we can enumerate all possible choices of Par_z on S_z. It's easy to calculate the corresponding Dep_z, Ret_z, and Anc_z functions, as well as the retrieval cost ρ_z and storage cost $\sigma(\mathcal{T}_z, \rho_z)$ for each choice of Par_z. These are all the feasible states.

4.4.3.3. Recurrence on separators. On a separator z with child c, because $S_z \subseteq S_c$, the feasible states on z are just restrictions of those on c:

$$\sigma(\mathcal{T}_z, \rho_z) = \min\{\sigma(\mathcal{T}_c, \rho_z) : \mathcal{T}_c\big|_{S_z} = \mathcal{T}_z\}$$

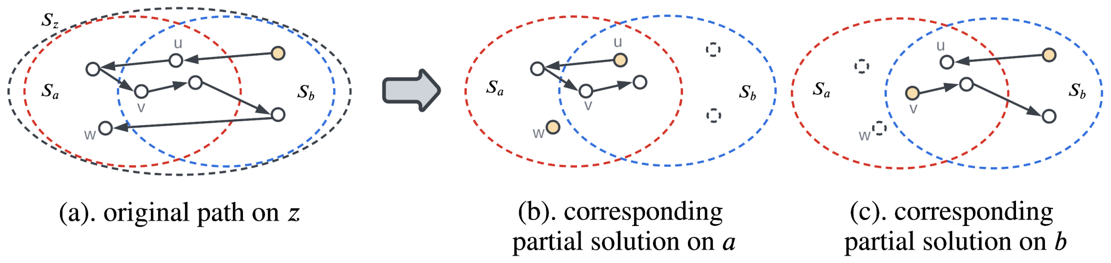
where $\mathcal{T}_c\big|_{S_z}$ is the natural restriction of $\mathcal{T}_c$ on S_z:

$$\mathcal{T}_c\big|_{S_z} = \Big(\mathrm{Par}_c\big|_{S_z}, \mathrm{Dep}_c\big|_{S_z}, \mathrm{Ret}_c\big|_{S_z}, \mathrm{Anc}_c\big|_{S_z}\Big)$$

where $\mathrm{Anc}_c\big|_{S_z}$ is Anc_c restricted on S_z with the additional requirement that its output is also an intersection with S_z. (So that the range of Anc_z is 2^{S_z}, as in the definition.)

4.4.3.4. Recurrence on joins. Suppose we are at a join z with children a, b, where $S_z = S_a \cup S_b$.

Compatibility. Naturally, we want to find all partial solutions $(\mathcal{T}_a, \rho_a)$ and $(\mathcal{T}_b, \rho_b)$ that "combine" into a given $(\mathcal{T}_z, \rho_z)$, and then take the minimum storage over the objective of all such combinations. The first attempt is to consider the states $\mathcal{T}_a, \mathcal{T}_b$ to be $\mathcal{T}_z\big|_{S_a}$ and $\mathcal{T}_z\big|_{S_b}$.



S_z
S_a
S_b
u
v
w
(a). original path on *z*
(b). corresponding partial solution on *a*
(c). corresponding partial solution on *b*

Figure 4.8. Illustration for compatibility. A node is colored if it is materialized.

However, in $\mathcal{T}_z$ there could be $v \in S_a$ such that $\text{Par}_z(v) \in V_{[b]} \setminus S_a$, as in node u of Fig. 4.8. We call these the *uprooted nodes*. To resolve problems like this, we need a more detailed definition of which $(\mathcal{T}_a, \mathcal{T}_b)$ can "combine" into $\mathcal{T}_z$. Let COMPATIBILITY (Algorithm 12) be a function which, given $\mathcal{T}_z, \mathcal{T}_a, \mathcal{T}_b$, returns a boolean value indicating whether $(\mathcal{T}_a, \mathcal{T}_b)$ are compatible with $\mathcal{T}_z$. We say $(\mathcal{T}_a, \mathcal{T}_b)$ is *compatible* with $\mathcal{T}_z$ in this case.

The Un-Uprooting process. The first step in COMPATIBILITY is to resolve the aforementioned problem of uprooted nodes. To do this, we first call SCAN UPROOTED NODES (Algorithm 10) to get the two sets of uprooted nodes U_a, U_b for $\mathcal{T}_a, \mathcal{T}_b$ respectively. It's not hard to see that U_a is just the nodes $v \in S_a$ such that $\text{Par}_z(v) \notin V_{[a]}$.

Afterwards, we apply UN-UPROOT (Algorithm 11) to loop through S_z topologically and calculate the correct $\text{Par}, \text{Ret}, \text{Anc}$ functions for both $\mathcal{T}_a$ and $\mathcal{T}_b$. This process reverses the idea of "uprooting" described in Section 4.4.1.

Fig. 4.9 gives a demonstration of how UN-UPROOT obtain these functions for $\mathcal{T}_a$. If $(\text{Par}_z(v), v)$ is as case 1, then v is un-uprooted (materialized), and we modify $\text{Anc}_a(v)$ and $\text{Ret}_a(v)$ accordingly. If $(\text{Par}_z(v), v)$ is as case 2, then we calculate $\text{Anc}_a(v)$ and $\text{Ret}_a(v)$ based on v's parent $\text{Par}_z(v)$. If $(\text{Par}_z(v), v)$ is as case 3, we subtract $\text{Dep}_a(v)$ from the dependency counts of all ancestors of v, including v itself. We note that case 4 is not dealt with in this step.

Distributing dependency. The next step in COMPATIBILITY is to check whether the functions $\text{Dep}_a, \text{Dep}_b$ are compatible with Dep_z. Specifically, nodes in S_a, S_b could have

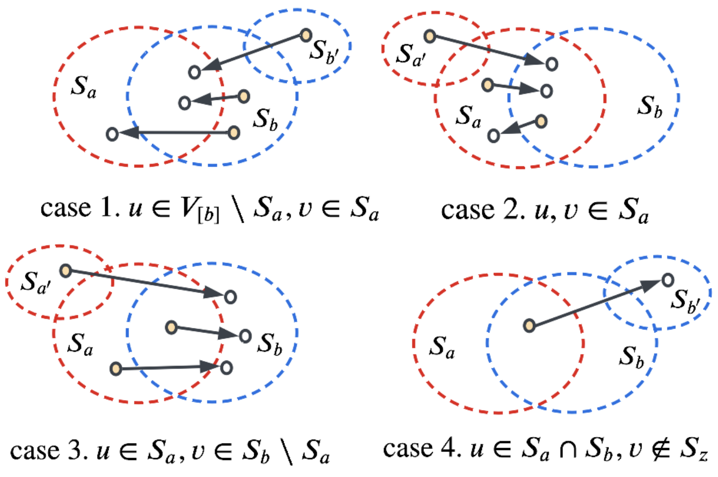

S_a
S_b
$S_{b'}$
case 1. $u \in V_{[b]} \setminus S_a, v \in S_a$
$S_{a'}$
S_a
S_b
case 2. $u, v \in S_a$
$S_{a'}$
S_a
S_b
case 3. $u \in S_a, v \in S_b \setminus S_a$
S_a
S_b
$S_{b'}$
case 4. $u \in S_a \cap S_b, v \notin S_z$

Figure 4.9. Four types of edge (u, v) involved when restricting $\mathcal{T}_z$ to $\mathcal{T}_a$.

external dependencies from $V_{[a]} \setminus S_a$ and $V_{[b]} \setminus S_b$, as in case 4 of Fig. 4.9. These external dependencies are from outside S_z, so they are untouched in the looping process in UNUPROOT. Consequently, we have to manually check whether the external dependencies in $\mathcal{T}_a$ and $\mathcal{T}_b$ adds up to that in $\mathcal{T}_z$, much like how we distribute dependency number k to the two children in case 4 of Fig. 4.7. The pseudo code for this calculation is inside COMPATIBILITY. (Algorithm 12)

We further note that we only need to distribute the external dependencies of nodes in $S_a \cap S_b$:

Lemma 4.21. *In a nice tree decomposition T, let $z \in V_T$ be a join with children a, b. If a' is a descendant of a in T, then $v \in S_{a'} \setminus S_a$ does not have neighbors in $S_b \setminus S_a$.*

Proof. For each edge (u, v) in G, there must be a bag containing both u and v. However, there is no bag containing both a node in $S_{a'} \setminus S_a$ and a node in $S_b \setminus S_a$, as a is in the unique path from a' to b. $\square$

Calculating ρ. Given that $(\mathcal{T}_a, \mathcal{T}_b)$ are compatible with $\mathcal{T}_z$, we want to find the objective, $\sigma(\mathcal{T}_z, \rho_z)$, with the recurrence relation involving $\sigma(\mathcal{T}_a, \rho_a) + \sigma(\mathcal{T}_b, \rho_b)$ for suitable ρ_a and ρ_b. However, we can't simply take $\rho_a + \rho_b = \rho_z$ due to the complicated procedure of combining

$\mathcal{T}_a, \mathcal{T}_b$ into $\mathcal{T}_z$. We thus implement Distribute Retrieval (Algorithm 13) to calculate ρ_Δ such that $\rho_a + \rho_b = \rho_z - \rho_\Delta$ and then iterate through all such ρ_a and ρ_b. A justification of this procedure can be found in Appendix C.2.2.

Recurrence relation. Finally, we have all we need for the recurrence relation:

$$\sigma(\mathcal{T}_z, \rho_z) = \min\left\{\sigma(\mathcal{T}_a, \rho_a) + \sigma(\mathcal{T}_b, \rho_b) - \mathit{uproot} - \mathit{overcount}\right\}$$

where the minimum is taken over all $(\mathcal{T}_a, \mathcal{T}_b)$ that are compatible with $\mathcal{T}_z$ and all $\rho_a + \rho_b = \rho_z - \rho_\Delta$, and

$$\mathit{uproot} = \sum_{v \in U_a} (s_v - s_{\mathrm{Par}_z(v),v}) + \sum_{v \in U_b} (s_v - s_{\mathrm{Par}_z(v),v}), \text{ and}$$

$$\mathit{overcount} = \sum_{v \in S_a \cap S_b} s_{\mathrm{Par}_z(v),v}.$$

If k is constant, then the recurrence relation takes $\mathrm{poly}(n)$ time. This is because there are $\mathrm{poly}(n)$ many possible states on S_a, S_b and S_z, and it takes $\mathrm{poly}(n)$ steps to check the compatibility of $(\mathcal{T}_a, \mathcal{T}_b)$ with $\mathcal{T}_z$ and compute ρ_Δ.

Output The minimum storage cost of a global solution is hence just $\min\{\sigma(\mathcal{T}_z), \rho_z\}$ over all states $\mathcal{T}_z$ and ρ_z, where z is the designated root of the nice tree decomposition.

We conclude this section with the following theorem.

Theorem 4.22. *For a constant $k \geq 1$, on the set of graphs whose undelying undirected graph has treewidth at most k,* MinSum Retrieval *admits an FPTAS, while* Bounded-Sum Retrieval *has an $(1, 1+\varepsilon)$ bi-criteria approximation that finishes in poly$(n, \frac{1}{\varepsilon})$ time.*

To see that our algorithm above is an FPTAS for MSR, the proof is almost identical to the proof of Theorem 4.18 (Section 4.4.1.3) once we note that the number of partial solutions on each z is $\mathrm{poly}(n)$.

An FPTAS for MMR arises from a similar procedure. When the objective becomes the maximum retrieval cost, we can use ρ_z to represent the maximum retrieval cost in the partial solution. We then modify $\mathrm{Dep}_z(v)$ to represent the highest retrieval cost among all the nodes that are dependent on v. The recurrence relation is also changed accordingly. One can note that, like before, the new tuple $\mathcal{T}_z$ contains all the information we need for a subsolution on $G_{[z]}$.

The same algorithms extend to $(1, 1+\varepsilon)$ bi-criteria approximation algorithms for BSR and BMR naturally, as the objective and constraint are reversed.

4.5. Experiments and Improved Heuristics for MSR and BMR

In this section, we propose three new heuristics that are inspired by empirical observations and theoretical results. We will discuss the experimental setup, datasets used, and experimental results for empirical validation of the performance of the algorithms. The performance and run time of these new algorithms are compared with previous best-performing heuristics.[13]

In all figures, the vertical axis (objective and run time) is presented in *logarithmic scale.* Run time is measured in *milliseconds.*

4.5.1. Datasets and Construction of Graphs

We use real-world GitHub repositories of varying sizes as datasets, from which we construct version graphs. Each commit corresponds to a node with its weight (storage cost) equal to its size in bytes. Between each pair of parent and child commits, we construct bidirectional edges. The storage and retrieval costs of the edges are calculated, in bytes, based on the actions (such as addition, deletion, and modification of files) required to change one version to the other in the direction of the edge. We use simple `diff` to calculate the deltas, hence

[13]Our code can be found at https://github.com/Sooooffia/Graph-Versioning.

the storage and retrieval costs are proportional to each other. Graphs generated this way are called "**natural graphs**" in the rest of the section.

In addition, we also aim to test (1) the cases where the retrieval and storage costs of an edge can greatly differ from each other, and (2) the effect of tree-like shapes of graphs on the performance of algorithms. Therefore, we also conduct experiments on modified graphs in the following two ways:

(1) **Random compression.** We simulate compression of data by scaling storage cost with a random factor between 0.3 and 1, and increasing the retrieval cost by 20% (for de-compression). In realistic cases, storage and retrieval costs for the "delta" between two versions are often proportional, but randomness is added for generality of our experiments.

(2) **ER construction.** Instead of the naturally constructing edges between each pair of parent-child commits, we construct the edges as in an Erdős-Rényi random graph: between each pair (u, v) of nodes, with probability p both deltas (u, v) and (v, u) are constructed, and with probability $1-p$ neither are constructed. This creates graphs much less tree-like than the natural construction. In particular, ER graphs have treewidth $\Theta(n)$ with high probability if the number of edges per vertex is greater than a small constant [Gao12].

The datasets we use are from GitHub's repositories, namely, `LeetCodeAnimation`,[14] `styleguide`,[15] `996.ICU`,[16] and `freeCodeCamp`.[17] The characteristic of these datasets can be found in Table 4.4.

We ran the experiments on a PC with an Intel i9-13900K and 64GB RAM. We used the python packages Networkx [HSSC08] and GitPython to generate version graphs. All algorithms were implemented using C++. To compute minimum spanning arborescence, we

[14] https://github.com/MisterBooo/LeetCodeAnimation
[15] https://github.com/google/styleguide
[16] https://github.com/996icu/996.ICU
[17] https://github.com/freeCodeCamp/freeCodeCamp

Dataset	#nodes	#edges	avg. materialization	avg. storage
datasharing	29	74	7672	395
styleguide	493	1250	1.4e6	8659
996.ICU	3189	9210	1.5e7	337038
freeCodeCamp	31270	71534	2.5e7	14800
LeetCodeAnimation	246	628	1.7e8	1.2e7
LeetCode 0.05	246	3032	1.7e8	1.0e8
LeetCode 0.2	246	11932	1.7e8	1.0e8
LeetCode 1	246	60270	1.7e8	1.0e8

Table 4.4. Natural and ER graphs overview.

applied Gabow et al.'s algorithm [GGST86], and we used Böther et al.'s code for implementation [BKW23].

4.6. Experiments and Improved Heuristics for MSR and BMR

In this section, we propose three new heuristics that are inspired by empirical observations and theoretical results. We will discuss the experimental setup, datasets used, and experimental results for empirical validation of the performance of the algorithms. The performance and run time of these new algorithms are compared with previous best-performing heuristics[18].

In all figures, the vertical axis (objective and run time) is presented in *logarithmic scale*. Run time is measured in *milliseconds*.

4.6.1. Datasets and Construction of Graphs

We use real-world GitHub repositories of varying sizes as datasets, from which we construct version graphs. Each commit corresponds to a node with its weight (storage cost) equal to its size in bytes. Between each pair of parent and child commits, we construct bidirectional edges. The storage and retrieval costs of the edges are calculated, in bytes, based on the actions (such as addition, deletion, and modification of files) required to change one version to the other in the direction of the edge. We use simple `diff` to calculate the deltas, hence

[18]Our code can be found at https://github.com/Sooooffia/Graph-Versioning.

the storage and retrieval costs are proportional to each other. Graphs generated this way are called “**natural graphs**” in the rest of the section.

In addition, we also aim to test (1) the cases where the retrieval and storage costs of an edge can greatly differ from each other, and (2) the effect of tree-like shapes of graphs on the performance of algorithms. Therefore, we also conduct experiments on modified graphs in the following two ways:

(1) **Random compression.** We simulate compression of data by scaling storage cost with a random factor between 0.3 and 1, and increasing the retrieval cost by 20% (for de-compression). In realistic cases, storage and retrieval costs for the “delta” between two versions are often proportional, but randomness is added for generality of our experiments.

(2) **ER construction.** Instead of the naturally constructing edges between each pair of parent-child commits, we construct the edges as in an Erdős-Rényi random graph: between each pair (u, v) of nodes, with probability p both deltas (u, v) and (v, u) are constructed, and with probability $1-p$ neither are constructed. This creates graphs much less tree-like than the natural construction. In particular, ER graphs have treewidth $\Theta(n)$ with high probability if the number of edges per vertex is greater than a small constant [Gao12].

4.6.2. Algorithms Implementation

4.6.2.1. Baselines. In considering MSR and BMR, we used LMG (refer to Section 4.2.1) and Modified Prim(MP) [BCH+15] as the respective baselines in assessing the performance of algorithms. Both are best-performing heuristics in previous experiments for the respective problems [BCH+15].

4.6.2.2. LMGA: improving LMG. For MSR, we implemented a modification for LMG named LMG-All, abbreviated LMGA below. (Algorithm 5) Instead of searching for the most efficient version to materialize per step, we can enlarge the scope of this search to explore the payoff of modifying any single edge.

Specifically, we will keep a set of active edges E_{active}, initialized to $E(G_{aux})$. On each iteration, let $\mathrm{Par}(v)$ be the parent of v in the current solution T. For all $e = (u, v) \in E_{active}$, we will consider the potential solution T_e, obtained by adding the edge e and removing $(\mathrm{Par}(v), v)$ from T. We then update T to T_{e^*} and remove e^* from E_{active}, where e^* maximizes $\rho_e = \frac{R(T)-R(T_e)}{S(T_e)-S(T)}$.

While LMGA considers more edges than LMG, it is not obvious that LMGA always provides a better solution. Due to its greedy nature, the first move might be better, but it may possibly be stuck in a worse local optimum.

Algorithm 5: LMG-ALL

```
   Input : Extended version graph $G_{aux}$, storage constraint $\mathcal{S}$
1  $T \leftarrow$ minimum arborescence of $G_{aux}$ rooted at $v_{aux}$ w.r.t. weight function $s$
2  Let $R(T)$ and $S(T)$ be the total retrieval and storage cost of $T$
3  Let $P(v)$ be the parent of $v$ in $T$
4  while $S(T) < \mathcal{S}$ do
5      $(\rho_{max}, (u_{max}, v_{max})) \leftarrow (0, \varnothing)$
6      for $e = (u, v) \in E$ where $u$ is not a descendant of $v$ in $T$ do
7          $T_e = T \setminus (P(v), v) \cup \{e\}$
8          if $R(T_e) > R(T)$ then
9              continue
10         else if $S(T_e) \leq S(T)$ then
11             $\rho_e \leftarrow \infty$
12         else
13             $\rho_e \leftarrow (R(T) - R(T_e))/(s_e - s_{P(v),v})$
14         if $\rho_e > \rho_{max}$ then
15             $\rho_{max} \leftarrow \rho_e$
16             $(u_{max}, v_{max}) \leftarrow e$
17     if $\rho_{max} = 0$ then
18         return $T$
19     $T \leftarrow T \setminus \{(P(v_{max}), v_{max})\} \cup \{(u_{max}, v_{max})\}$
20 return $T$
```

4.6.2.3. DP heuristics. We also propose DP heuristics on both MSR and BMR, as inspired by algorithms in Sections 4.3 and 4.4. Importantly, we note that DP algorithms proposed for bounded treewidth graphs involve the calculation of nice tree decompositions and complicated subroutines (UN-UPROOTING, etc.), which severely slow down the running time on graphs with high treewidths. To speed up the algorithm, we instead only run the DP on bi-directional trees (namely, with treewidth 1) extracted from our general input graphs, with the steps below:

(1) Calculate a minimum spanning arborescence A of the graph G rooted at the first commit v_1. We use the sum of retrieval and storage costs as weight.

(2) Generate a bidirectional tree G' from A. Namely, we have $(u, v), (v, u) \in E(G')$ for each edge $(u, v) \in E(A)$.

(3) Run the proposed DP for MSR and BMR on directed trees (see Section 4.4.1 and Section 4.3) with input G', and return the solution.

In addition, we also implement the following modifications for MSR to further speed up the algorithm:

(1) Total *storage* cost is discretized instead of retrieval cost, since the former generally has a smaller range.

(2) Geometric discretization is used instead of linear discretization.

(3) A pruning step is added, where the DP variable discards all subproblem solutions whose storage cost exceeds some bound.

All three original features are necessary in the proof for our theoretical results, but in practice, the modified implementations show comparable results but significantly improves the running time.

4.6.3. Results and Discussion

4.6.3.1. Results in MSR. Section 4.6.3.1, Fig. 4.11, and Fig. 4.12 demonstrate the performance of the three MSR algorithms on natural graphs, randomly compressed natural graphs, and random compression ER graphs. The running times for the algorithms are shown in

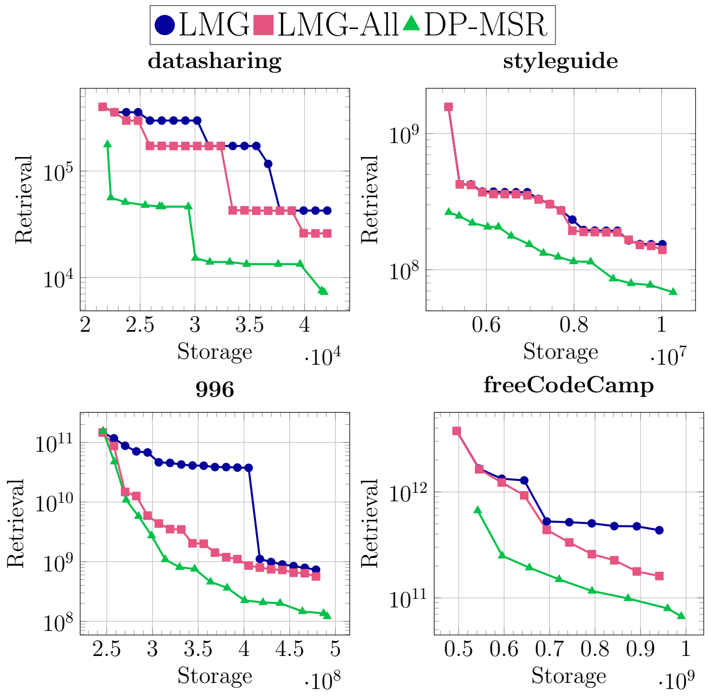


Figure 4.10. Performance of MSR algorithms on natural graphs. OPT is obtained by solving an integer linear program (ILP) using Gurobi [Gur22]. ILP takes too long to finish on all graphs except datasharing.

Fig. 4.11 and Fig. 4.12. Note since run time for most non-ER graphs exhibit similar trends, many are omitted here due to space constraint. Also note that, since all data points by the DP-MSR are generated with a single run of the DP, its running time is shown as a horizontal line over the full range for storage constraint.

We run DP-MSR with $\varepsilon = 0.05$ on most graphs, except $\varepsilon = 0.1$ for `freeCodeCamp` (for the feasibility of run time). The pruning value for DP variables is at twice the minimum storage for uncompressed graphs, and ten times the minimum storage for randomly compressed graphs.

Performance analysis. On most graphs, DP-MSR outperforms LMGA, which in turn outperforms LMG. This is especially clear on natural version graphs, where DP-MSR solutions are near 1000 times better than LMG solutions on 996.ICU. in Section 4.6.3.1. On datasharing, DP-MSR almost perfectly matches the optimal solution for all constraint ranges.

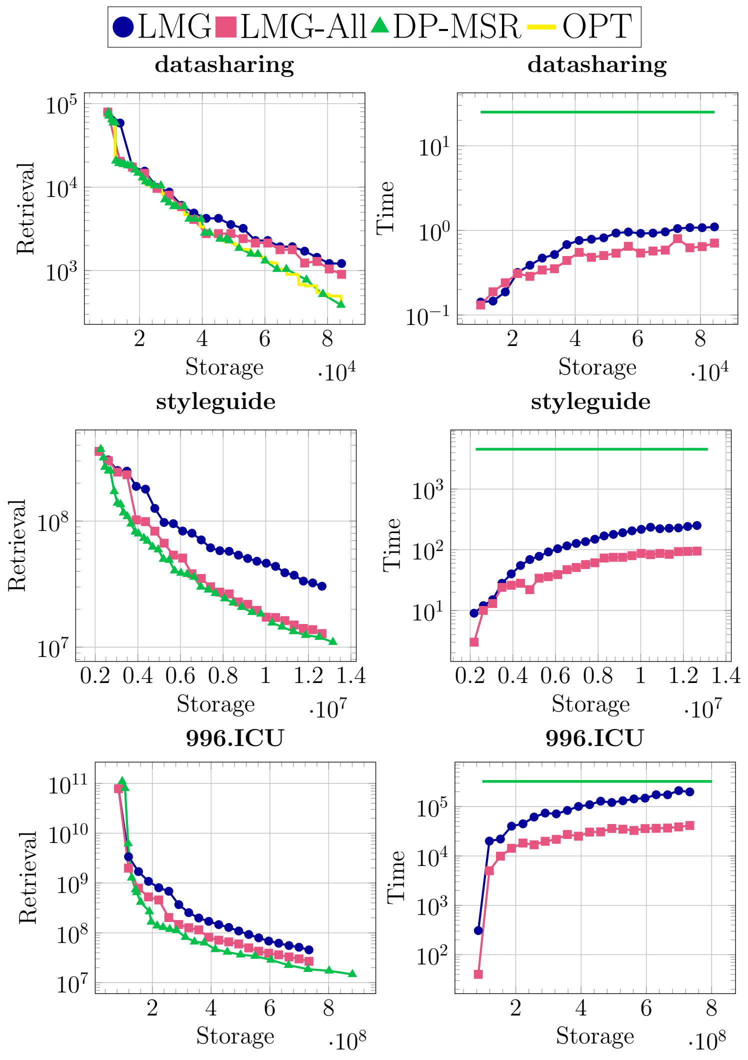


Figure 4.11. Performance and run time of MSR algorithms on compressed graphs.

On naturally constructed graphs (Section 4.6.3.1), LMGA often has comparable performance with LMG when storage constraint is low. This is possibly because both algorithms can only iterate a few times when the storage constraint is almost tight. DP-MSR, on the other hand, performs much better on natural graphs even for low storage constraint.

On graphs with simulated random compression (Fig. 4.11), the dominance of DP in performance over the other two algorithms become less significant. This is anticipated because of the fact that DP only runs on a subgraph of the input graph. Intuitively, most of the information is already contained in a minimum spanning tree when storage and retrieval

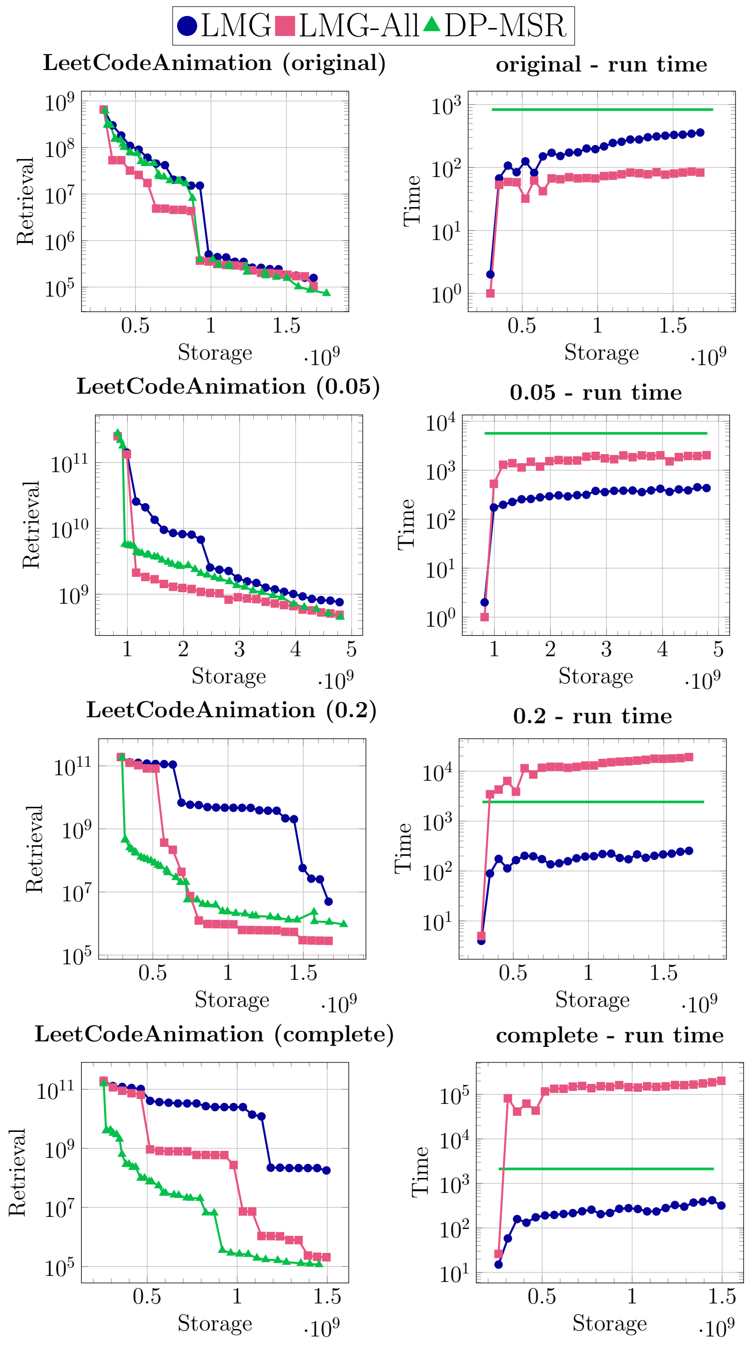


Figure 4.12. Performance and run time of MSR algorithms on compressed ER graphs.

costs are proportional. Otherwise, the dropped edges may be useful. (They could have large retrieval but small storage, and vice versa.)

Finally, LMG's performance relative to our new algorithms is much worse on ER graphs. This may be due to the fact that LMG cannot look at non-auxiliary edges once the minimum arborescence is initialized, and hence losing most of the information brought by the extra edges. (Fig. 4.12).

Run time analysis. For all natural graphs, we observe that LMGA uses no more time than LMG (as shown in Fig. 4.11). Moreover, LMGA is significantly quicker than LMG on large natural graphs, which was unexpected considering that the two algorithms have almost identical structures in implementation. Possibly, this can be due to the fact that LMG makes bigger, more expensive changes on each iteration (materializing a node with many dependencies, for instance) as compared to LMGA.

As expected, though, LMGA takes much more time than the other two algorithms on denser ER graphs (Fig. 4.12), due to the large number of edges.

DP-MSR is often slower than LMG, except when ran on the natural construction of large graphs (Fig. 4.11). However, unlike LMG and LMGA, the DP algorithm returns a whole spectrum of solutions at once, so it's difficult to make a direct comparison between the two. We also note that the runtime of DP heavily depends on the choice of ε and the storage pruning value. Hence, the user can trade-off the runtime with solution's quatlities by parameterize the algorithm with coarser configuration (i.e., higher ε).

4.6.3.2. Results in BMR. As compared to MSR algorithms, the performance and run time of our BMR algorithms are much more predictable and stable. They exhibit similar trends across different ways of graph construction as mentioned in earlier sections - including the non-tree-like ER graphs, surprisingly.

Due to space limitation, we present the results on natural graphs, as shown in Fig. 4.13, to respectively illustrate their performance and run time.

Performance analysis. For every graph we tested, DP-BMR outperforms MP on most of the retrieval constraint ranges. As the retrieval constraint increases, the gap between MP

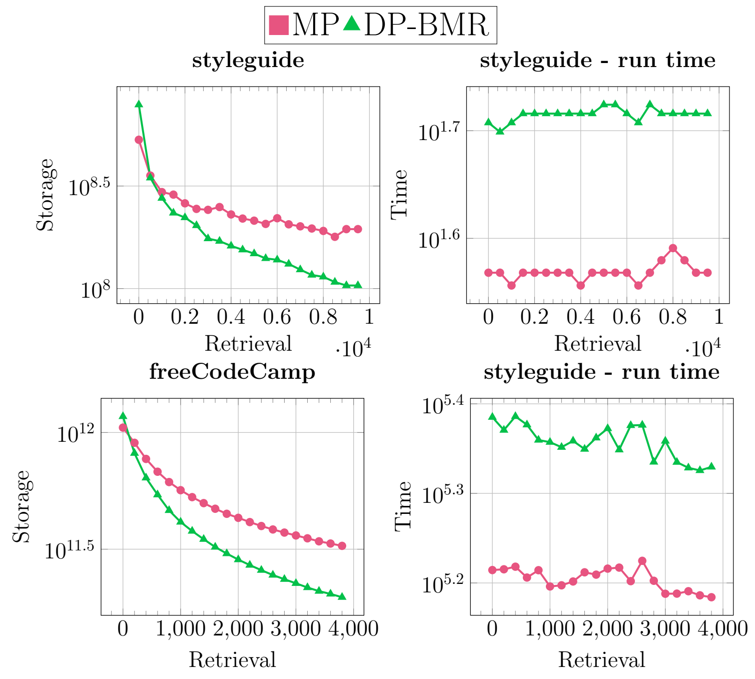


Figure 4.13. Performance and run time of BMR algorithms on natural version graphs.

and DP-BMR solution also increases. We also observe that DP-BMR performs worse than MP when the retrieval constraint is at zero. This is because the bidirectional tree have fewer edges than the original graph. (Recall that the same behavior happened for DP-MSR on compressed graphs)

We also note that, unlike MP, the objective value of DP-BMR solution monotonically decreases with respect to retrieval constraint. This is again expected since they are essentially optimal solutions the problem on the bidirectional tree.

Run time analysis. For all graphs, the runtimes of DP-BMR and MP are comparable within a constant factor. This is true with varying graph shapes and construction methods in all our experiments, and representative data is exhibited in Fig. 4.13. Unlike LMG and LMGA, their runtimes do not change much with varying constraint values.

4.6.3.3. Overall Evaluation. For MSR, we recommend always using one of LMGA and DP-MSR in place of LMG for practical use. On sparse graphs, LMGA dominates LMG both in performance and run time. DP-MSR can also provide a frontier of better solutions in a reasonable amount of time, regardless of the input.

For BMR, DP-BMR usually outperforms MP, except when the retrieval constraint is close to zero. Therefore, we recommend using DP in most situations.

4.7. Conclusion

In this paper, we developed fully polynomial time approximation algorithms for graphs with bounded treewidth. This often captures the typical manner in which edit operations are applied on versions. However, due to the high complexity of these algorithms, they are not yet practical to handle the size of real-life graphs. On the other hand, we extracted the idea behind this approach as well as previous LMG approach, and developed two heuristics which significantly improved both the performance and run time in experiments.

Future Works. There are many possible future directions. For one, a polynomial-time algorithm with bounded approximation ratio for general graph is desirable. Even for restricted classes of graphs, any development of a practical algorithm that can handle larger scale graphs is interesting. Moreover, in enterprise setting, often some versions are requested more often than others. It would be interesting to extend our work to handle such use cases.

4.8. Authors

This chapter was written by Anxin (Bob) Guo, Jingwei (Sofia) Li, Pattara Sukprasert, Samir Khuller, Amol Deshpande, Koyel Mukherjee. The preliminary version of this work is under submission at The International Conference on Very Large Data Bases (VLDB) 2023 [GLS+23].

CHAPTER 5

Densest Subgraph

The densest subgraph problem is a fundamental problem in graph mining that has been studied extensively for decades, both because of its theoretical challenges and its practical importance. The numerous applications of the problem include community detection and visualization in social networks [AHDBV05, GJL+13, JXRF09, KRRT99, CHKZ03, RTG14], motif discovery in protein and DNA [FNBB06, SSA+15, DHZ22, SHK+10], and pattern identification [DJD+09, AKS+14, HSB+16].

Significant effort has been made in the theoretical computer science community in computing exact and approximate densest subgraphs under various models of computation, in particular in the static [Cha00, KS09, BGP+20, CQT22, TG15, GGT89, Gol84], streaming [BHNT15], distributed [BGM14, SV20, GLM19], parallel [DBS18, BKV12, HQC22, DCS17], dynamic [BHNT15, SW20, CQ22, CHv+22], and privacy-preserving [FHS22, NV21, DLR+22] settings. However, despite a plethora of theoretical improvements on these fronts, there still does not exist practical near-optimal densest subgraph algorithms that can scale up to the largest publicly-available graphs with hundreds of billions of edges. In particular, for the largest such graphs, `hyperlink2012` (with roughly 113 billion edges) and `clueweb` (with roughly 37 billion edges), no previous approximations for the densest subgraph were known that are better than a 2-approximation.

There are two typical approaches for solving the densest subgraph problem exactly. The first is to solve a combinatorial optimization problem using a linear program solver. The other is to set up a flow network with size polynomial in the size of the original graph, binary search on the density, and then run a maximum flow algorithm on it. However, the caveat to both approaches is that they are not scalable to massive modern graphs; namely, both approaches have large polynomial runtimes and the best theoretical algorithms for these approaches are often not practical. Moreover, as the maximum flow problem is P-complete, there is no hope of fast parallel algorithm for the exact maximum flow problem [GSS82]. Because of this bottleneck, many have instead investigated approaches for approximate densest subgraphs. The best-known approximation algorithms for the densest subgraph problem fall into two categories.

The first category contains parallel approximation algorithms, which work by iteratively removing carefully chosen subsets of low-degree vertices while computing the density of the induced subgraph of the remaining vertices; then, the induced subgraph with the largest density is taken as the approximate densest subgraph [Cha00, BKV12, BHNT15] using $\text{poly}(\log n)$ rounds of peeling vertices with degree smaller than some threshold. Unfortunately, such methods give $(2+\varepsilon)$-approximations at best and no one has thus far made such methods work in $\text{poly}(\log n)$ rounds and give better approximations.

The second category consists of algorithms obtained from the *multiplicative weight update (MWU)* method. The multiplicative weight update framework approximately solves an optimization problem by using expert oracles to update the weights assigned to the variables multiplicatively and iteratively over several rounds depending on how the experts performed in previous rounds. The MWU framework allows for $(1+\varepsilon)$-approximate densest subgraphs in $\text{poly}(\log n)$ iterations; however, it requires more work per iteration to update the weights of the variables. As such, neither approach is particularly scalable to massive graphs.

In terms of practical solutions, Boob et al. [BGP$^+$20] present a fast, sequential, iterative peeling algorithm called `Greedy++` that combines peeling with the MWU framework. Chekuri et al. [CQT22] show that running `Greedy++` for $\Theta(\frac{\Delta \log n}{\rho^* \varepsilon^2})$ iterations results in a $(1+\varepsilon)$-approximation of the densest subgraph, where ρ^* is the density of the densest subgraph. However, `Greedy++` is not parallel, and does not take advantage of modern multi-core and multiprocessor architectures. Recently, Harb et al. [HQC22] proposed an iterative algorithm based on *projections* that solves a quadratic objective function with linear constraints derived from the dual of the densest subgraph linear program of Charikar [Cha00]. For a graph with m edges and maximum degree Δ, they prove that their algorithm converges to a $(1+\varepsilon)$-approximation in $O(\sqrt{m\Delta}/\varepsilon)$ iterations, where each iteration takes $O(m)$ work. However, their algorithm uses `Greedy++` as a subroutine, which is inherently sequential due to its iterative peeling nature, and hence they do not have strong bounds on depth (parallel time).

In our work, we design fast practical algorithms that simultaneously make use of parallelism as well as the closely related concept of the *k-core decomposition*. The k-core decomposition decomposes the graph into *k-cores* of different values of k. Within the induced subgraph of each k-core, each vertex has degree at least k. It is a well-known fact that the density of the densest subgraph is within a factor of 2 of the maximum core value. However, it is less clear how to make use of this fact in creating a scalable algorithm for the largest publicly-available graphs. In this chapter, we design a pruning framework that, combined with our parallel densest subgraph subroutines, results in both theoretical as well as practical improvements over the state-of-the-art with speed-ups of up to hundreds of times. The main idea of our framework is to iteratively prune the graph using lower bounds on the density of densest subgraph computed from our parallel densest subgraph subroutines, while preserving the densest subgraph.

The concept of using pruning to obtain a smaller subgraph from which to approximate the densest subgraph is also used in the work of Fang et al. [FYC$^+$19]. However, in their work, they only prune once using the maximum core number divided by two and then run maximum flow on the pruned graph. Furthermore, they do not consider *adaptive pruning* using any approximate densest subgraph approximation, which is the basis of our pruning framework. Compared to the single-shot approach of pruning in previous work, we introduce a *multi-shot* pruning approach in this paper that can be mixed and matched with new approximate densest subgraph algorithms in the future to obtain scalable and efficient algorithms from these baselines.

The main contributions in our work are three-fold:

- We design a pruning framework that can be combined with *any* parallel or sequential $(1+\varepsilon)$-approximate algorithm for densest subgraph to yield a scalable near-optimal, $(1+\varepsilon)$-approximate densest subgraph algorithm. Our algorithms adaptively prunes the graph over multiple iterations based on the best approximation of the densest subgraph in the current iteration.
- We give a new faster, sorting-based MWU iterative algorithm based on `Greedy++` [BGP+20] that achieves the same theoretical number of iterations as Chekuri et al. [CQT22] but is more amenable to parallelization. Experimentally, our parallel sorting-based algorithm outperforms our parallel peeling-based algorithm, as well as previous state-of-the-art benchmarks on most graphs. For instance, compared with the state-of-the-art parallel algorithm by Chekuri et al., we achieve up to 114x speedup on the same machine.
- Leveraging the scalability of our parallel algorithms, we provide a number of previously unknown graph statistics and graph mining results on the largest of today's publicly available graphs, `hyperlink2012` and `clueweb`, using commodity multicore machines.

5.1. Preliminaries

For an undirected unweighted graph $G = (V, E)$, we let $n = |V|$ and $m = |E|$. We let $\deg_G(v)$ be the degree of vertex v with respect to G. We define the ***density*** of G to be $\rho(G) = \frac{|E|}{|V|}$. The goal of the ***densest subgraph*** problem is to find a subgraph $S \subseteq G$, such that $\rho(S)$ is maximized. We will use S^* to denote a densest subgraph of G with density ρ^*.

A central structure we study is the k-core of an undirected graph. We now define k-core formally.

Symbol	Meaning
$G = (V, E)$	undirected unweighted graph
n, m	number of vertices, edges resp.
$\deg(v)$	current degree of vertex v
Δ	current maximum degree of graph
c_p	Peeling-complexity
$\rho(G)$	current density of graph G
ρ^*	optimal density of graph G
$\tilde{\rho}$	the best density found in our algorithms
$core(G, k)$	k-core of G
$core(v)$	core number of v
k_{max}	max. non-empty core number
$\ell(v)$	current load of vertex v

Table 5.1. Common notation used throughout the paper.

Definition 5.1 (k-core)**.** *A k-core $core(G, k)$ of G is defined to be a maximal subgraph $S \subseteq G$ such that $\deg_S(v) \geq k$ for any $v \in V(S)$.*

It is well known that to find $core(G, k)$, one can repeatedly *peel*[1] an arbitrary vertex v from G so long as $\deg_G(v) < k$. This process terminates when all remaining vertices have degree at least k, or the graph becomes empty. If the remaining graph is not empty, then it is the unique subgraph, $core(G, k)$. Next, we define the *coreness* or *core number* of a vertex v:

Definition 5.2 (Core number)**.** *For any vertex v, we let $core(v) = k$ if k is the maximum integer such that v is in $core(G, k)$.*

An easy modification of the peeling algorithm described above yields $core(v)$ for all vertices v. We call this peeling-based algorithm ***Coreness***. In this algorithm, we pick a vertex with minimum degree and peel it one at a time until there is no vertex left. Let D be a variable which represents the maximum degree of peeled vertices at the time we peel them. Initially, $D = 0$. Once v is about to be peeled, we set $D \leftarrow \max(D, deg_G(v))$. We then set $core(v) \leftarrow D$ and peel v from G. We refer to the ordering of vertices that we peel

[1]Throughout this paper, a vertex v is *peeled* from G means that v is deleted and G is now $G \setminus \{v\}$.

in this process as a ***degeneracy ordering*** of the graph, which is unique up to permuting vertices in the order with the same coreness.

We also use the following notion of c-approximate k-core, which can be computed more efficiently than exact k-core

Definition 5.3 (c-Approximate k-Core)**.** *A* c***-approximate*** k***-core*** *is a partition of vertices into layers, such that a vertex* v *is in approximate core* $\hat{k}(v)$ *denoted* $apxcore(G, \hat{k}(v))$ *only if* $\frac{k(v)}{c} \leq \hat{k}(v) \leq ck(v)$*, where* $k(v)$ *is the coreness of* v*.*

Later on, we will want to find an ordering that is similar to degeneracy ordering, but certain *loads* of vertices are also given as an input. Let $\ell(v)$ be *load* of v. At each step, we peel the vertex that minimizes the term $\ell(v) + \deg_G(v)$. Note that after v is peeled, the term $\deg_G(v')$ could be decreased, if v is adjacent to v', as $\deg_G(v')$ is an induced degree of the remaining part. The vanilla degeneracy ordering is the ordering we obtain by setting $\ell(v) = 0$ for all v. We overload the definition of degeneracy ordering and say that we find a degeneracy ordering with respect to loads ℓ.

Model Definitions. We analyze the theoretical efficiency of our parallel algorithms in the *work-depth* model [CLRS09, Jaj92]. In this model, the ***work*** is the total number of operations executed by the algorithm and the ***depth*** (parallel time) is the longest chain of sequential dependencies. We assume that concurrent reads and writes are supported in $O(1)$ work/depth. A ***work-efficient*** parallel algorithm is one with work that asymptotically matches the best-known sequential time complexity for the problem. We say that a bound holds ***with high probability (w.h.p.)*** if it holds with probability at least $1 - 1/n^c$ for any $c \geq 1$.

We use the following parallel primitives in our algorithms: ***ParFor***, ***SuffixSum***, ***Find-Max***, ***Bucketing***, and ***IntegerSorting***. Each primitive takes a sequence A of length n. ***ParFor*** is a parallel version of a for-loop that we use to apply a function f to each element in the sequence. If a function f takes $O(t)$ work and $O(d)$ depth, then ***ParFor*** takes $O(tn)$ work and $O(d)$ depth. ***SuffixSum*** returns a sequence B where $B[j] = \sum_{i=j}^{n} A[i]$. ***FindMax*** returns an element with maximum value among those in the sequence. The ***SuffixSum*** and ***FindMax*** primitives take $O(n)$ work and $O(\log n)$ depth. ***IntegerSorting*** returns a sequence in the sorted order (either ascending or descending order) according to integer keys. We use two different implementations of IntegerSorting: the first is an algorithm by Raman [Ram90] which takes $O(n \log\log n)$ expected work and $O(\log n)$ depth w.h.p., and the second is a folklore algorithm that takes $O(n/\varepsilon)$ work and $O(n^{\varepsilon})$ depth for $0 < \varepsilon < 1$ [Vis10]. The decision to use one of these two sorting algorithms depends on whether work or depth is more important. We state the complexity of our algorithm in both ways when necessary.

5.1.1. Pruning with Cores

In this section, we describe a pruning idea that takes an input graph G and outputs a subgraph H of G such that (1) H is typically smaller than G and (2) any densest subgraph S^* is in H.

We begin with a property that relates a graph's density and its vertices' degrees. The following lemma is folklore.

Lemma 5.4 (Folklore, (see, e.g., [CQT22]))**.** *For a graph G, if there is a vertex v with degree* $\deg(v) < \rho(G)$*, then $G' = G \setminus \{v\}$ is a graph with density $\rho(G') > \rho(G)$.*

Proof. Notice that

$$\begin{aligned}
\rho(G) &= \frac{|E(G')| + \deg(v)}{|V|} \\
&= \frac{|V|-1}{|V|} \cdot \frac{|E(G')|}{|V|-1} + \frac{\deg(v)}{|V|} \\
&= \frac{|V|-1}{|V|} \cdot \rho(G') + \frac{\deg(v)}{|V|} \\
&= x \cdot \rho(G') + (1-x) \cdot \deg(v),
\end{aligned}$$

for some real number $x \in (0,1)$. The last line can be viewed as a weighted average between $\rho(G')$ and $\deg(v)$. Since $\deg(v) < \rho(G)$, it has to be the case that $\rho(G') > \rho(G)$ so that their average becomes $\rho(G)$. $\square$

As a corollary, any vertex in a densest subgraph has induced degree at least $\rho(S^*)$.

Corollary 5.5. *Let S^* be the densest subgraph, then for any $v \in V(S^*)$, $\deg_G(v) \geq \deg_{S^*}(v) \geq \lceil \rho^* \rceil$.*

Intuitively, if we are given some $\tilde{\rho} < \rho^*$, we can keep removing any vertex v such that $v \leq \lceil \tilde{\rho} \rceil$ from the graph until no such vertex is left. By Corollary 5.5, no vertices from the densest subgraph S^* will be removed in this process. Notice that this process is very similar to the algorithm for computing $core(G, k)$ described in Section 5.1. In fact, we can relate k-core to the densest subgraph.

Lemma 5.6. *For some $k \leq \lceil \rho^* \rceil$, let $C = core(G, k)$ be a k-core of G. It must be the case that $S^* \subseteq C$.*

Proof. We prove this by contradiction. Assume that there exists some vertex $v \in S^* \setminus C$. Let $H = S^* \cup C$. Notice that, for any vertex $v \in S^* \cup C$, $\deg_H(v) \geq k$; if $v \in S^*$, then $\deg_H(v) \geq \deg_{S^*}(v) \geq k$ and if $v \in C$, then $\deg_H(v) \geq \deg_C(v) \geq k$. Hence, $S^* \cup C$ is a k-core with more vertices than C, implying that C is not maximal, which is a contradiction. $\square$

Moreover, the maximum non-empty core gives us a lower bound on ρ^*.

Corollary 5.7. *Let k_{max} be the maximum integer such that the core(G, k_{max}) is not empty. Let C be the $\lceil \frac{k_{max}}{2} \rceil$-core. Then $S^* \subseteq C$.*

Proof. For any v in $C = core(G, k_{max})$, we have $\deg_S(v) \geq k_{max}$. Hence,

$$\rho(C) = |E_C|/|V_C| \geq \frac{\sum_{v \in V_C} \deg_C(v)/2}{|V_C|} \geq k_{max}/2.$$

Hence, $\rho^* \geq \rho(C) \geq k_{max}/2$. It follows from Lemma 5.6 that S^* is contained in the $\lceil \frac{k_{max}}{2} \rceil$-core. □

Similarly, the largest non-empty c-approximate k-core, $\hat{k}_{max}$ also gives us a lower bound on ρ^*, in terms of the density of a (smaller) approximate core:

Corollary 5.8. *Let $\hat{k}_{max}$ be the maximum integer such that the apxcore$(G, \hat{k}_{max})$ is not empty. Let C be the $\lceil \frac{\hat{k}_{max}}{2c} \rceil$ approximate core. Then $S^* \subseteq C$.*

Proof. The proof is identical to that of Corollary 5.7; the only difference is that since the core is approximate, the lower bound on $\deg_S(v)$ for any v in $C = apxcore(G, \hat{k}_{max})$, is $\deg_S(v) \geq \hat{k}_{max}/c$. □

5.2. Pruning-and-Refining Framework

Based on the properties described in Section 5.1, any algorithm that yields a lower bound on ρ^* can be used for pruning the graph while retaining the densest subgraph. The main idea of the framework is as follows. Let L be a lower bound on ρ^*. We can prune the input graph G by computing G', which is an $\lceil L \rceil$-core of G and then search for the densest subgraph in G' instead of G. This process can be potentially repeated multiple times, making it useful in algorithms that iteratively refine increasingly tight lower bounds for ρ^* over a sequence of steps.

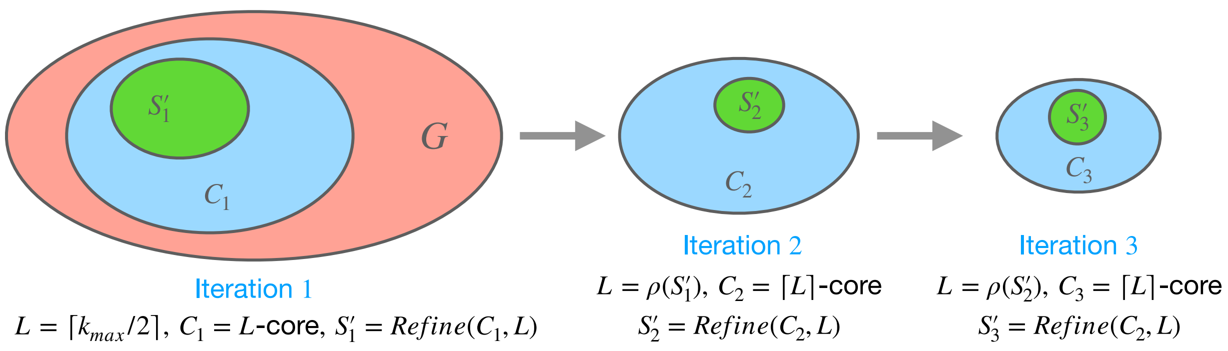

S_1'
C_1
G
S_2'
C_2
S_3'
C_3
Iteration 1
$L = \lceil k_{max}/2 \rceil$, $C_1 = L$-core, $S_1' = Refine(C_1, L)$
Iteration 2
$L = \rho(S_1')$, $C_2 = \lceil L \rceil$-core
$S_2' = Refine(C_2, L)$
Iteration 3
$L = \rho(S_2')$, $C_3 = \lceil L \rceil$-core
$S_3' = Refine(C_2, L)$

Figure 5.1. Example Illustrating the Pruning-and-Refining Framework (Algorithm 6). The i-th iteration of the algorithm computes a lower bound on the density, L, computes the $C_i = \lceil L \rceil$ core of G, and then applies an $Refine$ algorithm on C_i to compute a new subgraph S_i'. In the example, the density of each successive S_i' is increasing, and the cores C_i decrease in size.

To the best of our knowledge, the idea of using cores to prune the graph adaptively while refining the approximate densest subgraph solution has not been done in the literature. The closest idea is the `CoreExact` algorithm of Fang et al. [FYC+19]. Their pruning rules first compute ***Coreness***, and then inspect connected components from the $\lceil \frac{k_{max}}{2} \rceil$-core. They then take the maximum density found among the connected components as a lower bound for their binary searches (they use k_{max} as an upper bound) and run a flow-based algorithm on each connected component separately. Their pruning rules do not help much if there is only a single component in the $\lceil \frac{k_{max}}{2} \rceil$-core. On the other hand, our framework applies pruning for multiple iterations adaptively, even for graphs with a single connected component. Our framework can be seen as a generalized version of the algorithm of Fang et al.

5.2.1. Framework Overview

We apply this idea in an algorithmic framework for computing an approximate densest subgraph, which is shown in Algorithm 6. The pseudocode uses *exact pruning* i.e., uses the value of the exact k_{max}-core, but we also describe how to use the approximate k-cores in text below. On Lines 1–4, we compute the lower bound L by applying Corollary 5.7 and a k-core algorithm, or an approximate k-core algorithm. Both algorithms take linear

Algorithm 6: Pruning-and-Refining Framework

```
   Input  : an input graph $G = (V, E)$, number of iterations $T$, and an ordering
            function $O$
   Output: an approximate densest subgraph $S$
1  $cores, k_{max} \leftarrow Coreness(G)$
2  $G \leftarrow core(G, \lceil k_{max}/2 \rceil)$                    // From Corollary 5.5
3  $S \leftarrow G$                                                // Initial pruning
4  $L \leftarrow \lceil k_{max}/2 \rceil$                   // Trivial bound from $core(G, k_{max})$
5  for $i$=1 to $T$ do
6      $S' \leftarrow Refine(G, L)$                       // Refine candidate subgraph
7      if $\rho(S') > \rho(S)$ then
8          $S \leftarrow S'$
9          $L \leftarrow \max(L, \rho(S))$
10         $G \leftarrow core(G, \lceil L \rceil)$    // Additional pruning using $cores$ from Line 1
11 return $S$
```

$(O(m + n))$ work, but approximate k-core has provably poly-logarithmic depth. For exact k-core, we use the bucketing-based k-core implementation of [DBS17, DBS18], which takes $O(m+n)$ expected work and $O(c_p \log n)$ depth with high probability, where c_p is the peeling-complexity. ***Peeling-complexity*** is defined as the number of steps needed to completely peel the graph when each step peel *all vertices* with minimum degrees. For approximate k-core, we use the implementation of Liu et al. [LSY+22], which gives a $(2 + \delta)$-approximation to all core numbers and takes $O(m + n)$ expected work and $O(\log^3 n)$ depth whp. The lower bound L for the approximate k-core approach is computed using Corollary 5.8.

On Lines 5–10, we iterate for T rounds, where each round calls a function $Refine$ which attempts to compute a higher density subgraph. We then return the approximate densest subgraph found on Line 11. In Section 5.2.2 we describe various options for the $Refine$ function.

5.2.2. Refinement Algorithms

Next, we describe algorithms that can be used for the $Refine$ function in Algorithm 6. We first describe two existing sequential algorithms, the peeling algorithm and `Greedy++`, and then introduce our parallel algorithms.

Peeling Algorithm [Cha00]. At each step, we compute the density of the current graph. Then, we pick a vertex v with the minimum induced degree and remove it from the graph. We continue until there are no vertices remaining, and return the subgraph with the maximum density found in this process. The peeling algorithm can be parallelized [DBS17] but can have linear depth in the worst case. Charikar [Cha00] proves that the subgraph returned by this peeling algorithm has a density at least half the optimal density, i.e., it gives a 2-approximation to the densest subgraph.

`Greedy++` [BGP+20]. Algorithm 7 presents a greedy load-based densest subgraph algorithm, for which the state-of-the-art `Greedy++` algorithm is a special case. Initially, each vertex v is associated with a load $\ell(v) = 0$ (Lines 2–3). The algorithm runs for T iterations (Lines 4–11). On each iteration, we compute the degeneracy order O with respect to the load ℓ on graph H to obtain the ordered set of vertices $v_1, \ldots, v_n$ (Line 6). Then, on Lines 7–11, we peel vertices in this order. When v_i is peeled, we compare the density of the remaining subgraph to the density of the best subgraph found so far, and save the denser of the two. We also update the loads, setting $\ell(v_i) \leftarrow \ell(v_i) + \deg(v_i)$, where $\deg(v_i)$ here is the induced degree of v_i when it is peeled. We return the best subgraph found after $T = \Theta(\frac{\Delta \log n}{\rho^* \varepsilon^2})$ iterations. Recall that Δ is the maximum degree and $0 < \varepsilon < 1$ here is an adjustable parameter. This algorithm yields a $(1+\varepsilon)$-approximation of the densest subgraph as shown in [CQT22]. Note that the first iteration of Greedy++ is exactly the peeling algorithm of Charikar. The pseudocode for `Greedy++` is shown in Algorithm 7.

Algorithm 7: Greedy Load-Based Densest Subgraph

```
Input  : an input graph $G = (V, E)$, number of iterations $T$, ordering function $O$
Output: an approximate densest subgraph $S$
1  $S \leftarrow G$
2  for $v$ in $V$ do
3      $\ell(v) \leftarrow 0$
4  for $i$=1 to $T$ do
5      $H = (V_H, E_H) \leftarrow (V, E)$
6      let $v_1, \ldots, v_n$ be the ordering provided by the function $O$
7      for $j$=1 to $n$ do
8          if $\rho(H) > \rho(S)$ then
9              $S \leftarrow H$
10         $\ell(v_j) \leftarrow \ell(v_j) + \deg_H(v_j)$
11         $H \leftarrow H \setminus \{v_j\}$
12 return $S$
```

The original `Greedy++` algorithm is implemented in a way where the degeneracy ordering and the update steps are fused together. It will become clear once we introduce our algorithm below why we decouple these two steps.

`GreedySorting++` (our algorithm). Our second algorithm uses a simpler method for computing O, in that it orders vertices based on their loads at the *beginning* of the iteration. The motivation for this algorithm is that sorting is faster in practice than the iterative peeling process used in `Greedy++`, and is also highly parallel. Therefore, on Line 6, we compute $v_1, \ldots, v_n$, such that $\ell(v_1) \leq \ell(v_2) \leq \ldots \leq \ell(v_n)$. Because of the way we decouple the ordering and update steps in `Greedy++`, Line 6 is the only difference between two algorithms. Next, we argue that `GreedySorting++` has the same guarantees in terms of approximation and number of rounds compared to `Greedy++`.

Theorem 5.9. *For $T = \Theta\left(\frac{\Delta \log n}{\lambda^* \varepsilon^2}\right)$, `GreedySorting++` outputs a $(1 + \varepsilon)$-approximation to the densest subgraph problem.*

Proof. (Sketch) The proof follows almost immediately from Section 4 of [CQT22]. To prove that `Greedy++` works, they define a linear program where each variable is corresponding

Algorithm 8: Parallel Density and Load Computation

```
   Input  : an input graph $G = (V, E)$, an ordering $v_1, \dots v_n$, current loads $\ell(\cdot)$
   Output: updated loads $\ell(\cdot)$, best density found $\rho_{max}$
1  $A :=$ array of size $n$
2  parfor $e = (v_j, v_k)$ in $E$ do
3      $A[\min(j,k)] \leftarrow A[\min(j,k)] + 1$
4  $B := SuffixSum(A)$
5  parfor $i = 1$ to $n$ do
6      $B[i] \leftarrow B[i]/(n - i + 1)$
7  $\rho_{max} \leftarrow \max_i B[i]$ // maximum density in this iteration
8  parfor $v_i$ in $V$ do
9      $\ell(v_i) \leftarrow \ell(v_i) + A[i]$ // update loads
10 return $\ell, \rho_{max}$
```

to the peeling order. They then utilize the multiplicative weight update (MWU) framework on the linear program.[2] The oracle problem that we need to solve is to find a good ordering. Lemma 4.6 of [CQT22] shows that the ordering obtained with `Greedy++` is a *good approximate ordering*. The proof of Lemma 4.6 work for any order that satisfies the following property: $\ell(v_i) \leq \ell(v_j) + \Delta$ if $i < j$. This is true for the ordering used in `GreedySorting++`, where we sort by the initial load of the vertices. Hence, by plugging this ordering, all of the proofs in [CQT22] go through. □

Remark. In our experiments, `Greedy++` converges in many fewer iterations than indicated by Theorem 5.9. We suspect that the existing analysis is not tight.

5.2.3. Parallel Implementation

In this subsection, we present our parallelizations of `Greedy++` and `GreedySorting++`. We still run both algorithms for T iterations, one iteration at a time, and our aim is to achieve low depth within each iteration.

Parallelization of the common parts. We first describe how to parallelize all parts of Algorithm 7 except for Line 6. Let $v_1, \dots, v_n$ be an ordering of vertices in which we will peel.

[2]See, e.g., [AHK12] for a survey on this topic.

Notice that, on the i'th iteration of the for-loop on Line 7, the induced subgraph of G that we respect is $S_i = G(v_i, \dots, v_n)$. This holds for all $1 \leq i \leq n$. For any edge $e = (v_j, v_k)$, e will contribute to the density of S_i if and only if $i \leq j$ and $i \leq k$.

Our implementation for computing the densities and updating the loads in parallel is shown in Algorithm 8. We first initialize an empty array A of size n (Line 1). Then, for each edge $e = (v_j, v_k)$, we add 1 to $A[\min(j,k)]$ (Lines 2–3). Let B be the suffix sum array of A (Line 4). It is the case that after the suffix sum, $B[i]$ corresponds to the number of edges remaining in the graph after vertices $v_1, \dots, v_{i-1}$ are peeled. To see why it is the case, let us consider a graph remains after $v_1, \dots, v_{i-1}$ are peeled, which is corresponding to a subgraph induced by $v_i, \dots, v_n$. Consider an edge $e = (v_j, v_k)$. e will appear in this subgraph if and only if both v_j and v_k are not yet peeled, i.e., $i \leq j$ and $i \leq k$. When we consider adding 1 to $A[\min(j,k)]$, e is accounted for in any subgraphs of the form $v_{i \leq \min(j,k)}, \dots, v_n$. We then compute the densities and take the maximum density on Lines 5–7. We update the loads in parallel on Lines 8–9. The work and the depth of this implementation are $O(n+m)$ and $O(\log n)$, respectively. As we described in Section 5.1, ***ParFor***, ***SuffixSum***, and ***FindMax*** all take linear work. ***SuffixSum*** and ***FindMax*** have $O(\log n)$ depth, and ***ParFor*** have $O(1)$ depth, so the depth bound follows.

`ParallelGreedy++`. In order to parallelize `Greedy++`, what is left for us is to parallelize the computation of the degeneracy ordering, which can be computed with a k-core algorithm in $O(m+n)$ expected work and $O(c_p \log n)$ depth with high probability. Notice that, when we peel all vertices with minimum degree at the same time, the degeneracy ordering can be different from the order obtained sequentially. The reason is that when a vertex is peeled in the sequential algorithm, it affects its neighbors degrees immediately. However, in the parallel version, this effect is delayed until the end of the peeling step where multiple vertices may be peeled together. We claim that this does not significantly affect the order. Consider a pair of vertices v_i and v_j. To make the proof in Theorem 5.9 go through, it suffices to show that

$i < j$ implies $\ell(v_i) \leq \ell(v_j) + \Delta$. We prove this by contrapositive. Suppose $\ell(v_i) > \ell(v_j) + \Delta$. Because the numbers of neighbors of both v_i and v_j are bounded by Δ, it is the case that, even if all of v_i neighbors are peeled and non of v_j neighbors are peeled, v_i will still be peeled after v_j, implying that $i > j$. As the invariant that $\ell(v_i) \leq \ell(v_j) + \Delta$ if $i < j$ holds true, the proof in Theorem 5.9 will go through.

Theorem 5.10. *For $T = \Theta\left(\frac{\Delta \log n}{\rho^* \varepsilon^2}\right)$, `ParallelGreedy++` outputs a $(1+\varepsilon)$-approximation to the densest subgraph problem. Moreover, each iteration takes $O(n+m)$ expected work and $O(c_p \log n)$ depth with high probability.*

`ParallelGreedySorting++`. We replace the degeneracy order in `ParallelGreedy++` with integer sorting to obtain `ParallelGreedySorting++`. As discussed in 5.1, integer sorting takes $O(n \log\log n)$ expected work and $O(\log n)$ depth w.h.p., or $O(n/\varepsilon)$ work and $O(n^\varepsilon)$ depth for $0 < \varepsilon < 1$. We have argued earlier that sorting does not affect the approximation guarantee of the algorithm. Therefore, we have the following theorem.

Theorem 5.11. *For $T = \Theta\left(\frac{\Delta \log n}{\rho^* \varepsilon^2}\right)$, `ParallelGreedySorting++` outputs a $(1+\varepsilon)$-approximation to the densest subgraph problem. Moreover, each iteration takes either $O(n \log\log n + m)$ expected work and $O(\log n)$ depth w.h.p. or $O(n+m)$ work and $O(n^\varepsilon)$ depth for $0 < \varepsilon < 1$.*

Combining Refinement with Pruning. Using the framework in Algorithm 6, we can combine a pruning method with *one iteration* of `Greedy++`, `GreedySorting++`, `ParallelGreedy++`, and `ParallelGreedySorting++` as the $Refine$ function. For example, we can combine approximate k-core for pruning with one iteration of `ParallelGreedySorting++`, which would give an algorithm with either $O(T(n \log\log n + m))$ expected work and $O(T \log n + \log^3 n)$ depth or $O(T(n+m))$ work and $O(T \log n + n^\varepsilon)$ depth for $0 < \varepsilon < 1$.

Note. While pruning gives speedups in practice, it does not improve the theoretical complexity of the algorithm, as there exists a graph where the core that we prune down to covers most of the original graph. As an example, the maximum non-empty core of a clique is the clique itself.

5.3. Experiments

In this section, we implement and benchmark different instantiations of the Pruning-and-Refining framework on real-world datasets. We also compare our algorithms with existing algorithms. One goal is to show that our approach is practical and is scalable. Another goal is to provide statistical data on large-scale graphs, in particular the densities of near-optimal densest subgraphs of these graphs, which has not been reported in the literature for some of the larger graphs.

Implementations. We implement `Greedy++`, `GreedySorting++`, and their parallel instantiations as our refinement algorithms. We consider two algorithms for pruning: pruning using exact k-cores, and pruning using approximate k-cores. Both pruning algorithms are parallel and are modular across all of the refinement algorithms. We use the exact k-core algorithm from Dhulipala et al. [DBS17] and the approximate k-core algorithm from Liu et al. [LSY+22]. We name our algorithms as follows:

(1) `PaRGreedy++`: is `ParallelGreedy++` combined with our Pruning-and-Refining framework.
(2) `PaRSorting++`: is `ParallelGreedySorting++` combined with our Pruning-and-Refining framework.

Our framework and all of the implementations will be made publicly available in the final version of this paper.

Existing Algorithms. We compare with the state-of-the-art algorithms from Fang et al. [FYC+19] (`CoreExact` and `CoreApp`), Boob et al. [BGP+20] (`Greedy++`), and Harb et al. [HQC22] (`FISTA`, `Frank-Wolfe`, `MWU`).

Fang et al. [FYC+19] implemented a variant of a maximum-flow algorithm to find the densest subgraph. Their `CoreExact` implementation is exactly the flow algorithm with binary search over the density. They perform a density lower-bound estimation using cores, which we described in more detail in Section 5.1.1. If a graph has many connected components, then a subgraph with maximum density lies exclusively in one component. Hence, they run the flow algorithm on each connected component separately. The densities of cores are then used to determine the lower bounds and upper bounds of the binary search needed for the flow computation. Their approximation algorithm, `CoreApp`, is an algorithm that finds $core(G, k_{max})$ directly. Once the maximum core is found, they return the component with the highest density. This algorithm yields a 2-approximation for our problem.

Harb et al. [HQC22] propose a gradient descent based algorithm called `FISTA`, where the number of iterations needed is $O(\sqrt{\Delta m}/\varepsilon)$. They use accelerated proximal gradient descent which is faster than the standard gradient descent approach [Nes83, BT09]. The output in each iteration is a feasible solution to a linear program for the densest subgraph problem. They then use `Greedy++`-inspired rounding, which they call fractional peeling, to round the linear program solution into an integral solution.

Setup. We use `c2-standard-60` Google Cloud instances (3.1 GHz Intel Xeon Cascade Lake CPUs with a total of 30 cores with two-way hyper-threading, and 236 GiB RAM) and `m1-megamem-96` Google Cloud instances (2.0 GHz Intel Xeon Skylake CPUs with a total of 48 cores with two-way hyper-threading, and 1433.6 GB RAM). We use hyper-threading in our parallel experiments by default. Our programs are written in C++. We use parallel primitives from the `GBBS` [DBS18] and `Parlay` [BAD20] libraries. The source code is compiled using g++ (version 10) with the -O3 flag. We terminate experiments that take over 2 hours. Using enough threads, with our framework, `Greedy++`, `GreedySorting++`, and their parallel versions finished within 2 hours for all of our experiments. However, `FISTA` took longer than 2 hours on all of the large graphs (`clueweb`, `twitter`, `friendster`, `hyperlink2012`), so we omit the entries for these datasets.

Datasets. We run our experiments on various synthetic and real-world datasets. The real-world datasets are obtained from SNAP [LK14] (`cahepth`, `ascaida`, `hepph`, `dblp`, `wiki`, `youtube`, `stackoverflow`, `livejournal`, `orkut`, `twitter`, `friendster`), Network Repository [RA15] (`brain`), Lemur project at CMU [BV04] (`clueweb`), and WebDataCommons [MVLB15] (`hyperlink2012`). The `hyperlink2012` graph is the largest publicly-available real-world graph today. `closecliques` is a synthetic dataset designed to be challenging for `Greedy++` [BGP+20, CQT22, HQC22]. We remove self-loop and zero-degree vertices from all graphs, and symmetrize any directed graphs. The sizes of our inputs and their maximum core values are shown in Table 5.2. We run most of our experiments on `c2-standard-60` machines. However, on the larger graphs (namely, `twitter`, `friendster`, `clueweb` and, `hyperlink2012`), we use `m1-megamem-96` machines as more memory is required.

Overview of Results. We show the following experimental results in this section.

- Our pruning strategy is very efficient as edges are 40× fewer on average and vertices are 1951× fewer on average.
- Our algorithms, similar to the state of the art, take only a few iterations to converge. `PaRSorting++`takes more iterations in particular, but it still converges to < 1.01-approximation within 10–20 iterations.
- When measuring wall-clock time, our algorithms outperform existing algorithms by a large margin.
- Our algorithms are highly parallelizable, achieving up to 29x self-relative parallel speedup on a 30-core machine with two-way hyperthreading.
- We measure empirical "width", which is a parameter that correlates to the number of iterations needed to converge. We observe that the empirical width is much smaller than the upper bound used to analyze the algorithm. This may lead to a more fine-grained analysis of many MWU-inspired algorithms.

Graph Dataset	Original Graph Num. Vertices	Original Graph Num. Edges	k_{max}	$core(G, \lceil \frac{k_{max}}{2} \rceil)$ Num. Vertices	$core(G, \lceil \frac{k_{max}}{2} \rceil)$ Num. Edges	Vertex Ratio	Edge Ratio
`closecliques`	3,230	95,400	59	3,230	95,400	1.000	1.000
`cahepth`	9,877	25,973	31	96	2,306	0.0097	0.088
`ascaida`	26,475	106,762	22	208	6,244	0.007	0.048
`hepph`	28,094	3,148,447	410	6,304	1,562,818	0.224	0.494
`dblp`	317,080	1,049,866	101	280	13,609	0.001	0.010
`brain`	784,262	267,844,669	1,200	187,494	137,354,946	0.239	0.512
`wiki`	1,094,018	2,787,967	124	3,807	344,553	0.034	0.090
`youtube`	1,138,499	2,990,443	51	12,836	439,678	0.011	0.110
`stackoverflow`	2,584,164	28,183,518	163	41,651	5,709,796	0.016	0.187
`livejournal`	4,846,609	42,851,237	329	6,090	1,054,941	0.001	0.022
`orkut`	3,072,441	117,185,083	253	71,507	13,469,722	0.023	0.113
`twitter`	41,652,230	1,202,513,046	2,484	24,480	36,136,023	0.001	0.029
`friendster`	65,608,366	1,806,067,135	304	1,474,236	271,902,207	0.022	0.146
`clueweb`	978,408,098	37,372,179,311	4,244	91,874	132,549,663	9.39e-05	0.003
`hyperlink2012`	3,563,602,789	112,920,331,616	10,565	250,477	1,046,929,322	7.02e-05	0.009

Table 5.2. Graph sizes, their maximum core values (k_{max}), their $\lceil k_{max}/2 \rceil$-core sizes, and their vertex (edge) ratios, where this quantity is the number of vertices (edges) in $core(G, \lceil \frac{k_{max}}{2} \rceil)$ divided by the number of vertices (edges) in G.

5.3.1. Core-Based Pruning

Pruning with $core(G, \lceil \frac{k_{max}}{2} \rceil)$. We first study the benefit of performing pruning using the *exact* k-core computation. The data for this experiment across all graphs is shown in Table 5.2. For most real-world graphs, the cores contain between 5× to 100× fewer edges than the actual graphs (40× fewer on average), and between 4× to 10000× fewer vertices (1951× fewer on average). The only exception is the `brain` dataset, where the core is half the actual size. Even in this case, the number of vertices left in the core is around 25% of the original graph. For the synthetic dataset `closecliques`, the input is designed so that the maximum-core is identical to the original graph, so there is no benefit to pruning. We note that this situation is very unlikely to occur in real-world datasets.

Due to the significant reduction in graph sizes in terms of both the number of vertices and number of edges, using $core(G, \lceil \frac{k_{max}}{2} \rceil)$ is almost always preferable over using G, especially since computing all cores of G (a linear-work algorithm, with reasonably high parallelism in practice [DBS17]) is inexpensive compared to the cost of running any of the refinement algorithms, which mostly require super-linear work. To summarize, we find that pruning is nearly always beneficial and should be applied prior to refinement.

Pruning with highest cores. As our algorithms progress, we perform additional pruning to shrink the graph even further. We include a table that has the sizes of the final graphs in our supplementary materials. In many cases, the sizes of the final graph we process are less than half of their $\lceil\frac{k_{max}}{2}\rceil$-cores. The extreme case here is `friendster`, where the final size is only .73 percent of the original size. Another extreme case is `clueweb`, where the best density found is roughly $\lceil\frac{k_{max}}{2}\rceil$, so there is little benefit from additional pruning. Across all datasets, we find that iterative pruning yields a 5.5× reduction in the number of vertices in the input subgraph to the refinement step, and a 16.3× reduction in the number of edges when comparing these quantities in $core(G, \lceil k_{max}/2\rceil)$ and $core(G, \lceil\tilde{\rho}\rceil)$, where $\tilde{\rho}$ is best density found by our algorithms.

5.3.2. Number of Iterations Versus Density

Next, we study the progress that different refinement algorithms make in our framework toward finding the maximum density. We select following algorithms for this experiment: `PaRGreedy++`, `PaRSorting++`, `FISTA`, `Greedy++`, `FrankWolfe`, and `MWU`. All algorithms were run for 20 iterations. The result is illustrated in Section 5.3.4. In fact, most algorithms converge really early. Two algorithms, namely, `PaRSorting++` and `MWU` take more iterations in many graphs. This correlates with our understanding of the *width* of MWU as we will discuss further below.

Graph Dataset	$\tilde{\rho}$	FISTA	MWU	FrankWolfe	Greedy++	PaRGreedy++	PaRSorting++
hepph	265.969	1.014707341	1.000240687	1.000327213	1.000067682	1.000105283	1.002712634
dblp	56.56522	1.000000354	1.000000354	1.000000354	1.000000354	1.000000046	1.000000046
brain	1057.458	1.000111601	1.00160832	1.001845553	1	1.000000204	1.002204238
wiki	108.5877	1	1	1.023273149	1	1	1.000070379
youtube	45.59877	1.000225276	1.00379449	1.045663475	1.000201142	1.000190795	1.00170595
stackoverflow	181.5867	1.000003855	1.000064436	1.014093922	1.000003855	1.000000082	1.000093301
livejournal	229.8459	1.002415697	1.017048758	1.13958293	1	1.000001356	1.022995862
orkut	227.874	1	1.000351195	1.003593795	1	1.000000112	1.000325247
twitter	1643.301	n/a	n/a	n/a	1.000000609	1	1.00267877
friendster	273.5187	n/a	n/a	n/a	n/a	1	1.000002608
clueweb	2122.5	n/a	n/a	n/a	n/a	1.000000222	1.000000222
hyperlinks	6496.649	n/a	n/a	n/a	n/a	1.000002839	1.066697279

Table 5.3. Approximation Ratio at the 10th iteration for various algorithms.

Graph Dataset	$\tilde{\rho}$	FISTA	MWU	FrankWolfe	Greedy++	PaRGreedy++	PaRSorting++
hepph	265.969	1.000425042	1.000105287	1.000109047	1.00000376	1.000001679	1.000307467
dblp	56.56522	1.000000354	1.000000354	1.000000354	1.000000354	1.000000046	1.000000046
brain	1057.458	1.000111601	1.000045394	1.000310274	1	1.000000163	1.000262295
wiki	108.5877	1	1	1.007213617	1	1	1
youtube	45.59877	1.000225276	1.001042122	1.015216864	1.000065138	1	1.000791673
stackoverflow	181.5867	1.000003855	1.000009362	1.003102886	1.000003855	1.000000082	1.000019569
livejournal	229.8459	1.001132038	1.000030021	1.036714839	1	1.000000124	1.000194885
orkut	227.874	1	1.000109722	1.001225862	1	1.000000065	1.000255332
twitter	1643.301	n/a	n/a	n/a	1.000000609	1	1.000029835
friendster	273.5187	n/a	n/a	n/a	n/a	1	1.000002608
clueweb	2122.5	n/a	n/a	n/a	n/a	1.000000222	1.000000222
hyperlinks	6496.649	n/a	n/a	n/a	n/a	1	1

Table 5.4. Approximation Ratio at the 20th iteration for various algorithms.

5.3.3. Approximation Ratio

In Table 5.3, we compare the densities returned from various algorithms at iteration 10 with the best density known. Excepts for `brain`, `twitter`, `friendster`, `clueweb`, and `hyperlinks`, the best density known is optimal.[3]

Except for `FrankWolfe`, all algorithms have approximation ratios less than 1.02 after 10 iterations. We also include a table that compares densities after iteration 20 in our supplementary materials. After 20 iterations, most approximation ratios are less than 1.001 Table 5.4. Among our datasets, `livejournal` is the graph with the highest approximation

[3]To compute the optimal density, we run a linear program solver on $core(G, \lceil\tilde{\rho}\rceil)$.

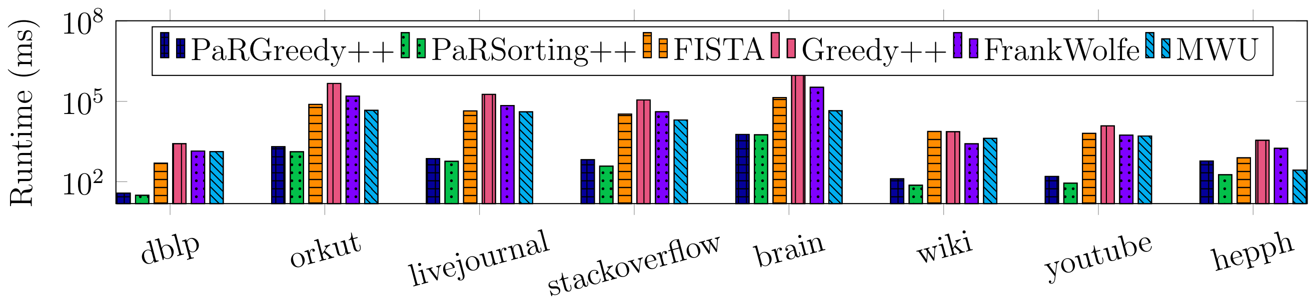


Figure 5.2. Running times of different densest subgraph algorithms on our small graph inputs.

ratios after 10 and 20 iterations. Note that `Greedy++`, `PaRGreedy++`, and `PaRSorting++` took the fewest iterations to converge.

5.3.4. Empirical Widths

As mentioned at the end of Section 5.2.2, in the multiplicative weight update framework, *"width"* is a parameter that is correlated to the number of rounds needed for a solution to converge. In our context, the width corresponds to the *maximum increase of a load* of a single vertex in any iteration.[4] Clearly the width is bounded by the maximum degree Δ, which is reflected in the $T = O(\frac{\Delta \log n}{\rho^* \varepsilon^2})$ iterations needed for our algorithms in the worst case.

However, this bound does not reflect reality as `Greedy++` usually converges in just a few iterations. Here, we partially explain this phenomenon by measuring the width empirically. Table 5.5 shows information about the width across multiple datasets gathered from our experiments. We observe that widths for running `PaRGreedy++` are much closer to the best density found, while widths from `PaRSorting++` are closer to Δ, meaning that `PaRSorting++` should take more iterations to converge. This supports what we observed in our experiments, and also may explain why it takes very few iterations, e.g., fewer than 10–20 iterations, for `PaRGreedy++` to converge.

[4]See, e.g., [CQT22, AHK12] for more details on width and this analysis.

			Max. Degree Δ			`PaRGreedy++`		`PaRSorting++`	
Graph Dataset	Num. Vertices	$\tilde{\rho}$	G	$\lceil \frac{k_{max}}{2} \rceil$-core	$\lceil \tilde{\rho} \rceil$-core	No Pruning	Pruning	No Pruning	Pruning
`closecliques`	3,230	29.55665	2,000	2,000	2,000	59	59	2,000	2,000
`cahepth`	9,877	15.5	65	31	31	31	31	65	31
`ascaida`	26,475	17.53409	2,628	146	76	44	45	2,628	56
`hepph`	28094	265.969	4,909	3,259	2,589	683	633	4,909	2,543
`dblp`	317,080	56.56522	343	114	114	114	114	343	114
`brain`	784,262	1,057.458	21,743	16,151	7,686	2,545	2,676	21,743	13,523
`wiki`	1,094,018	108.5877	141,951	2,288	1,025	312	338	141,951	1,023
`youtube`	1,138,499	45.59877	28,754	4,064	1,108	137	139	28,754	1,392
`stackoverflow`	2,584,164	181.5867	44,065	14,613	3,783	640	619	44,065	3,787
`livejournal`	4,846,609	229.8459	20,333	1,010	629	546	535	20,333	1,010
`orkut`	3,072,441	227.874	33,313	13,162	7,186	755	779	33,313	7,447
`twitter`	41,652,230	1,643.301	2,997,487	19,549	10,705	3,859	3,737	2,997,487	10,286
`friendster`	65,608,366	273.5187	5,214	2,952	2,190	1,092	1,018	5,214	2,431
`clueweb`	978,408,098	2,122.5	75,611,696	7,707	7,707	6,661	7,065	75,611,696	7,707
`hyperlink2012`	3,563,602,789	6,496.649	95,041,164	67,920	3,740	n/a	17,287	95,041,163	67,792

Table 5.5. Empirical widths in our experiments.

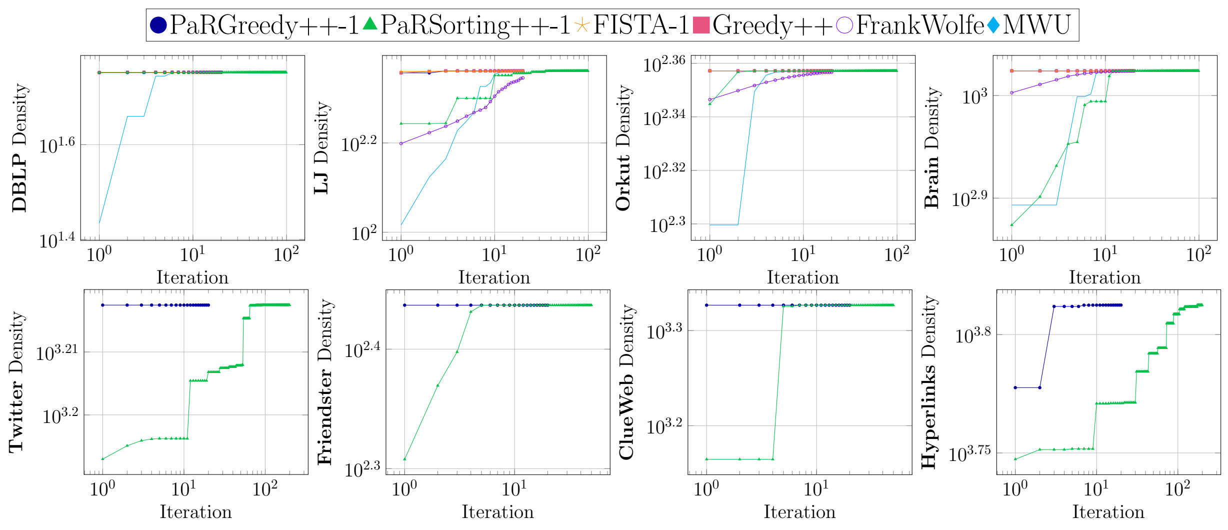


Figure 5.3. Densities on different iterations for various algorithms. Only our algorithms can successfully process large graphs (bottom row) within the 2 hours limit.

In Section 5.3.4, we show the progress between densities and running times on various algorithms. Note that one of our algorithms, either `PaRGreedy++` or `PaRSorting++` is the fastest to converge in wall-clock time.

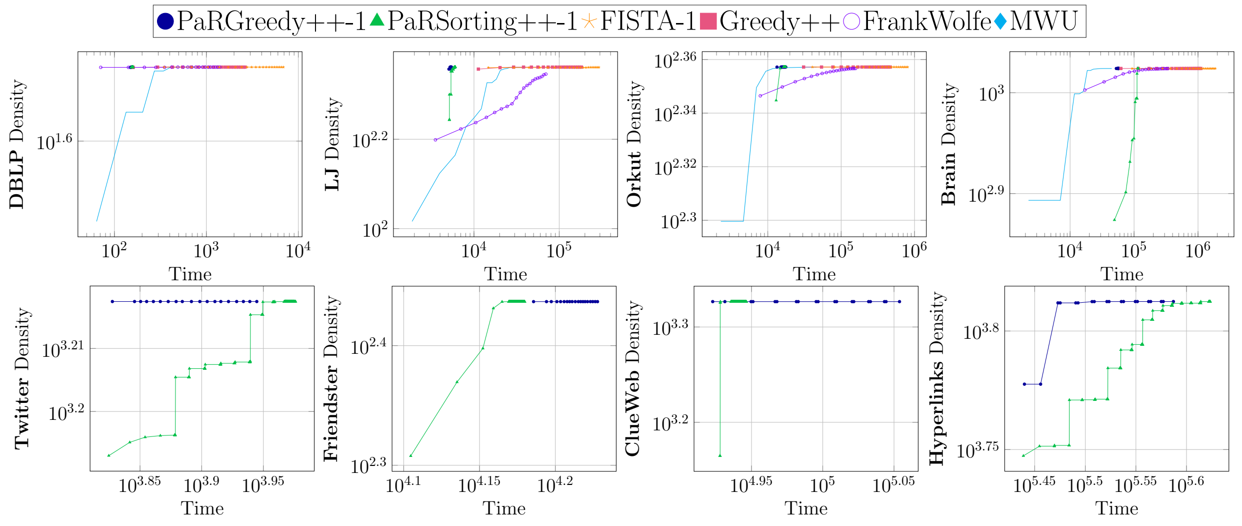


Figure 5.4. Densities and time (ms) for various algorithms. `PaRGreedy++`, `PaRSorting++`, `FISTA` are run on one thread here for smaller graphs (top row) and 60 threads for the large graphs (bottom row).

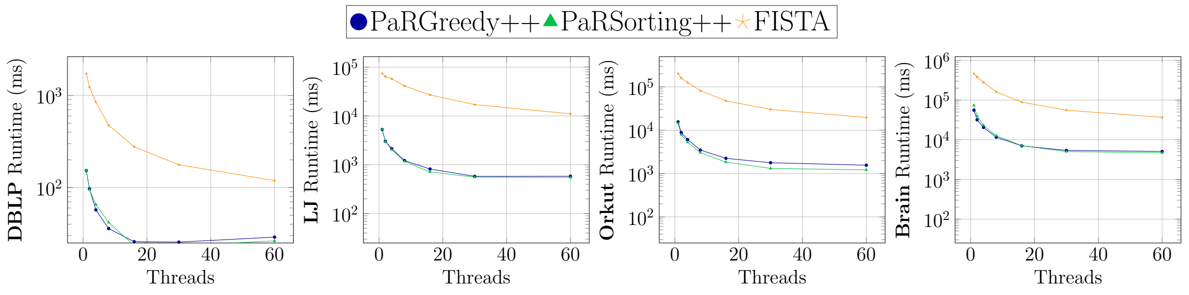


Figure 5.5. Runtime (ms) of `PaRGreedy++`, `PaRSorting++`, and FISTA versus the number of threads when running for 5 iterations.

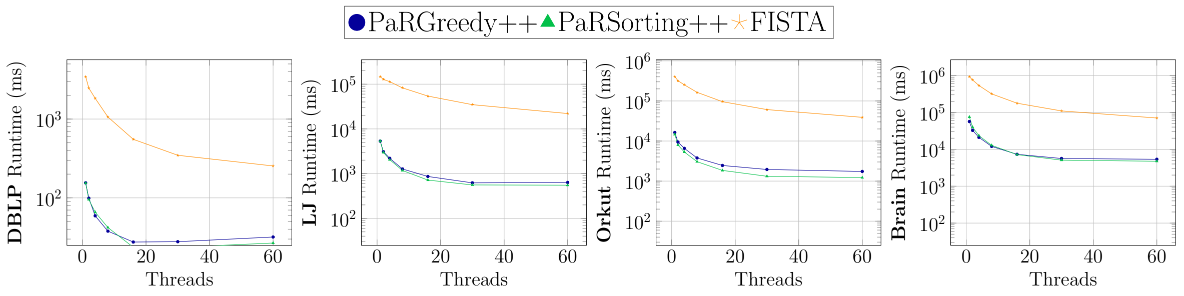


Figure 5.6. Runtime (ms) of `PaRGreedy++`, `PaRSorting++`, and FISTA versus the number of threads when running for 10 iterations.

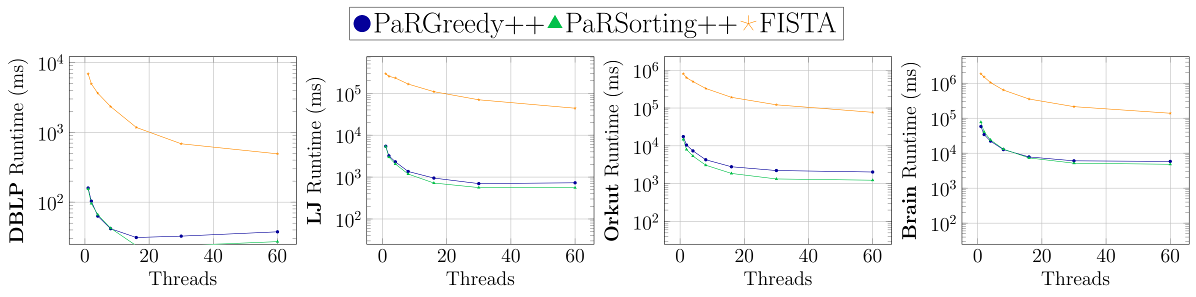


Figure 5.7. Runtime (ms) of `PaRGreedy++`, `PaRSorting++`, and FISTA versus the number of threads when running for 20 iterations.

5.3.5. Impact of Parallelism

Here we show the scalability of our algorithms compared to the parallel version of FISTA in Fig. 5.5. Fig. 5.5 shows the running time of the algorithms (in milliseconds) versus the number of threads used by our algorithms and parallel FISTA when each algorithm is run for 5 iterations. We show additional plots in our supplementary materials. We see that our `PaRSorting++` algorithm achieves greater self-relative speedups than FISTA and `PaRGreedy++`. Specifically, `PaRSorting++` achieves up to a $29.64x$ self-relative speedup on `livejournal` while FISTA achieves up to a $14x$ self-relative speedup on `dblp` and `PaRGreedy++` achieves up to a $5.51x$ self-relative speedup on `orkut`. Furthermore, both of our algorithms take shorter time than parallel FISTA regardless of the number of threads.

5.3.6. Comparing the Total Running Time

We first compare `PaRSorting++` with the algorithms given in [FYC+19] (see Table 5.6). We ran experiments on two of their algorithms, namely, `CoreExact` and `CoreApp`. `CoreExact` took too long to run on most datasets. `CoreApp` is faster than our implementation, however since it uses $core(G, k_{max})$, this algorithm gives a 2-approximation and is less accurate than our algorithms.

We include plots that compare the total runtime of our algorithms (PaRGreedy++, PaRSorting++) with Greedy++ [BGP+20], FISTA, FrankWolfe, and MWU [HQC22] in our supplementary materials. In short, when measuring the quality of our solutions in running time, our algorithms outperform all existing algorithms by significant margins (3–40× even on one thread situation). Moreover, even when using multi-threading, our algorithms are the only algorithms that finish processing large graphs (twitter, friendster, clueweb, and hyperlinks) within 2 hours. Other algorithms could not finish it even if we run them for 5 hours.

		CoreApp		PaRSorting++		CoreExact
Graph Dataset	$\tilde{\rho}$	Density	time(ms)	Density	time(ms)	time(ms)
asCaida	17.53	16.72	4704	17.53	13	4704
caHepTh	15.5	15.5	417	15.5	6	417
brain	1057.45	1006.94	23573	1057.45	76959	n/a
dblp	56.57	56.5	65	56.57	154	247018
hepph	265.97	205	61	265.97	1351	n/a
lj	229.85	195.86	2295	229.85	5344	n/a
orkut	227.87	219.32	17757	227.87	14740	n/a
stackoverflow	181.59	173.23	3599	181.59	4462	n/a
wiki	108.59	102.12	408	108.59	487	n/a
youtube	45.60	43.03	273	45.60	642	n/a

Table 5.6. Comparison between PaRSorting++ on one thread and algorithms from [FYC+19].

5.4. Conclusion

We have introduced a framework that combines pruning and refinement for solving the approximate densest subgraph problem. We have designed new parallel variants of the sequential Greedy++ algorithm, and achieved state-of-the-art performance by plugging them into our framework. We have shown that our algorithms can scale to the large hyperlinks2012 and clueweb graphs for the first time in the literature.

5.5. Authors

This chapter was written by Pattara Sukprasert, Quanquan Liu, Laxman Dhulipala, and Julian Shun.

The preliminary version of this work is under submission at ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD) 2023 [SLDS23].

CHAPTER 6

Conclusion

In this thesis, we study four problems related to finding good subgraphs. In the kECSS problem, we propose an algorithm that finds a $(1 + \varepsilon)$-approximate fractional solution in near-linear time. We also show a rounding method that rounds the solution obtained into a $(2 + \varepsilon)$-approximate integral solution. For Dynamic Spanner, we close the recourse gap between an oblivious adversary setting and an adaptive adversary setting. We show that an algorithm that achieves both good recourse bounds and good running time is attainable at least for 3-spanner. For Dataset versioning problem, we propose Dynamic Programming algorithms for trees and extend them to work with bounded tree-width graph. We also make use of our algorithm to show better bounds, compared to best known heuristic. For Densest subgraph problem, we propose a parallel provable algorithm that works very well in practice. We manage to scale our implementation so that it runs on `hyperlinks2012`, which is one of largest available graphs within a reasonable time.

Despite recent progress, many questions remain open. For kECSS, the natural open problem is to improve the running time of other problems in the survivable network design family where the demand between pairs of vertices are not necessarily uniform. Jain's framework [Jai01] gives 2-approximation to the whole family. Hence, the goal is to develop efficient algorithms with similar approximation ratios. For dynamic spanners, it would be interesting to see faster algorithms when for maintaining spanners with stretches higher than 3. For Dataset versioning and related problems in directed graphs, not much is known on the algorithmic side. Hence, it would be interesting to see any bi-criteria approximation algorithm that works for general graphs. Lastly, for densest subgraph, there are many exciting directions. It is interesting to see a provable condition in which we can surely shrink the graph with our pruning rule. Here, we might have to look beyond worst-case analysis. As we have an efficient algorithm for approximate dense subgraphs, it is interesting to obtain an efficient exact algorithm. Despite the development in network flows, especially the near-linear time algorithm by Chen et al. [CKL$^+$22], we do not have a good exact algorithm that works well in practice yet. The faster maximum flow result suggests such a possibility. In this direction, an efficient, possibly parallel, algorithm that solves the linear program for solving exact densest subgraph [Cha00] is also interesting.

References

[ABED$^+$21] Noga Alon, Omri Ben-Eliezer, Yuval Dagan, Shay Moran, Moni Naor, and Eylon Yogev. Adversarial laws of large numbers and optimal regret in online classification. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 447–455, 2021.

[ABL$^+$20] Chen Avin, Marcin Bienkowski, Andreas Loukas, Maciej Pacut, and Stefan Schmid. Dynamic balanced graph partitioning. *SIAM Journal on Discrete Mathematics*, 34(3):1791–1812, 2020.

[ABS$^+$20] Abu Reyan Ahmed, Greg Bodwin, Faryad Darabi Sahneh, Keaton Hamm, Mohammad Javad Latifi Jebelli, Stephen G. Kobourov, and Richard Spence. Graph spanners: A tutorial review. *Comput. Sci. Rev.*, 37:100253, 2020.

[ACP87] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic Discrete Methods*, 8(2):277–284, 1987.

[ADD$^+$93a] Ingo Althöfer, Gautam Das, David P. Dobkin, Deborah Joseph, and José Soares. On sparse spanners of weighted graphs. *Discrete & Computational Geometry*, 9:81–100, 1993.

[ADD$^+$93b] Ingo Althöfer, Gautam Das, David P. Dobkin, Deborah Joseph, and José Soares. On sparse spanners of weighted graphs. *Discret. Comput. Geom.*, 9:81–100, 1993.

[Adj18] David Adjiashvili. Beating approximation factor two for weighted tree augmentation with bounded costs. *ACM Transactions on Algorithms (TALG)*, 15(2):1–26, 2018.

[ADK+16] Ittai Abraham, David Durfee, Ioannis Koutis, Sebastian Krinninger, and Richard Peng. On fully dynamic graph sparsifiers. In *FOCS*, pages 335–344, 2016.

[AFI06] Giorgio Ausiello, Paolo Giulio Franciosa, and Giuseppe F. Italiano. Small stretch spanners on dynamic graphs. *J. Graph Algorithms Appl.*, 10(2):365–385, 2006. Announced at ESA'05.

[AH77] Kenneth Appel and Wolfgang Haken. The solution of the four-color-map problem. *Scientific American*, 237(4):108–121, 1977.

[AHDBV05] J. Ignacio Alvarez-Hamelin, Luca Dall'Asta, Alain Barrat, and Alessandro Vespignani. Large scale networks fingerprinting and visualization using the k-core decomposition. In *Proceedings of the 18th International Conference on Neural Information Processing Systems*, 2005.

[AHK12] Sanjeev Arora, Elad Hazan, and Satyen Kale. The multiplicative weights update method: a meta-algorithm and applications. *Theory of Computing*, 8(1):121–164, 2012.

[AKS+14] Albert Angel, Nick Koudas, Nikos Sarkas, Divesh Srivastava, Michael Svendsen, and Srikanta Tirthapura. Dense subgraph maintenance under streaming edge weight updates for real-time story identification. *VLDB J.*, 23(2):175–199, 2014.

[AKT21] Amir Abboud, Robert Krauthgamer, and Ohad Trabelsi. APMF < APSP ? gomory-hu tree for unweighted graphs in almost-quadratic time. *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1135–1146, 2021.

[Arc00] Aaron Archer. Inapproximability of the asymmetric facility location and k-median problems. 2000.

[Arc01] Aaron Archer. Two o(log*k)-approximation algorithms for the asymmetric k-center problem. In Karen Aardal and Bert Gerards, editors, *Integer Programming and Combinatorial Optimization*, pages 1–14, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[BAD20] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. ParlayLib - a toolkit for parallel algorithms on shared-memory multicore machines. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, page 507–509, 2020.

[BB73] Umberto Bertelè and Francesco Brioschi. On non-serial dynamic programming. *Journal of Combinatorial Theory, Series A*, 14(2):137–148, 1973.

[BBC$^+$14] Anant Bhardwaj, Souvik Bhattacherjee, Amit Chavan, Amol Deshpande, Aaron J Elmore, Samuel Madden, and Aditya G Parameswaran. DataHub: Collaborative Data Science & Dataset Version Management at Scale. *arXiv*, 2014.

[BBG$^+$20] Aaron Bernstein, Jan van den Brand, Maximilian Probst Gutenberg, Danupon Nanongkai, Thatchaphol Saranurak, Aaron Sidford, and He Sun. Fully-dynamic graph sparsifiers against an adaptive adversary. *arXiv preprint arXiv:2004.08432*, 2020.

[BBN19] Dan Brickley, Matthew Burgess, and Natasha Noy. Google Dataset Search: Building a search engine for datasets in an open Web ecosystem. *The World Wide Web Conference*, pages 1365–1375, 2019.

[BC16] Aaron Bernstein and Shiri Chechik. Deterministic decremental single source shortest paths: beyond the $o(mn)$ bound. In *STOC*, pages 389–397, 2016.

[BC17] Aaron Bernstein and Shiri Chechik. Deterministic partially dynamic single source shortest paths for sparse graphs. In *SODA*, pages 453–469, 2017.

[BC18] Aaron Bernstein and Shiri Chechik. Incremental topological sort and cycle detection in expected total time. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 21–34, 2018.

[BCCK16] Surender Baswana, Shreejit Ray Chaudhury, Keerti Choudhary, and Shahbaz Khan. Dynamic dfs in undirected graphs: breaking the o (m) barrier. In *Proceedings of the twenty-seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 730–739. SIAM, 2016.

[BCH$^+$15] Souvik Bhattacherjee, Amit Chavan, Silu Huang, Amol Deshpande, and Aditya G. Parameswaran. Principles of dataset versioning: Exploring the recreation/storage tradeoff. *Proc. VLDB Endow.*, 8(12):1346–1357, 2015.

[BDH$^+$19] Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, Cliff Stein, and Madhu Sudan. Fully dynamic maximal independent set with polylogarithmic update time. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 382–405, 2019.

[BDT14] Glencora Borradaile, Erik D Demaine, and Siamak Tazari. Polynomial-time approximation schemes for subset-connectivity problems in bounded-genus graphs. *Algorithmica*, 68(2):287–311, 2014.

[BEJWY20] Omri Ben-Eliezer, Rajesh Jayaram, David P Woodruff, and Eylon Yogev. A framework for adversarially robust streaming algorithms. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI symposium on principles of database systems*, pages 63–80, 2020.

[Ber17] Aaron Bernstein. Deterministic partially dynamic single source shortest paths in weighted graphs. In *ICALP*, volume 80, pages 44:1–44:14, 2017.

[BF21] Mahdi Belbasi and Martin Fürer. Finding all leftmost separators of size $\leq k$. In *Combinatorial Optimization and Applications: 15th International Conference, COCOA 2021, Tianjin, China, December 17–19, 2021, Proceedings*, page 273–287, Berlin, Heidelberg, 2021. Springer-Verlag.

[BFH19] Aaron Bernstein, Sebastian Forster, and Monika Henzinger. A deamortization approach for dynamic spanner and dynamic maximal matching. In *SODA*, pages 1899–1918, 2019.

[BFPK20] Alex Bogatu, Alvaro A. A. Fernandes, Norman W. Paton, and Nikolaos Konstantinou. Dataset Discovery in Data Lakes. *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 00:709–720, 2020.

[BG07] André Berger and Michelangelo Grigni. Minimum weight 2-edge-connected spanning subgraphs in planar graphs. In *International Colloquium on Automata, Languages, and Programming*, pages 90–101. Springer, 2007.

[BGM14] Bahman Bahmani, Ashish Goel, and Kamesh Munagala. Efficient primal-dual graph algorithms for MapReduce. In *International Workshop on Algorithms and Models for the Web Graph (WAW)*, volume 8882, pages 59–78, 2014.

[BGP$^+$20] Digvijay Boob, Yu Gao, Richard Peng, Saurabh Sawlani, Charalampos Tsourakakis, Di Wang, and Junxing Wang. Flowless: Extracting densest subgraphs without flow computations. In *Proceedings of The Web Conference 2020*, page 573–583, 2020.

[BGS21] Aaron Bernstein, Maximilian Probst Gutenberg, and Thatchaphol Saranurak. Deterministic decremental sssp and approximate min-cost flow in almost-linear time. *arXiv preprint arXiv:2101.07149*, 2021.

[BGW21] Sayan Bhattacharya, Fabrizio Grandoni, and David Wajc. Online edge coloring algorithms via the nibble method. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2830–2842. SIAM, 2021.

[BHI15] Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. Deterministic fully dynamic data structures for vertex cover and matching. In *SODA*, 2015.

[BHM$^+$21] Vladimir Braverman, Avinatan Hassidim, Yossi Matias, Mariano Schain, Sandeep Silwal, and Samson Zhou. Adversarial robustness of streaming algorithms through importance sampling. *arXiv preprint arXiv:2106.14952*, 2021.

[BHN16] Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. New deterministic approximation algorithms for fully dynamic matching. In *STOC*, pages 398–411, 2016.

[BHN17] Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. Fully dynamic approximate maximum matching and minimum vertex cover in $O(\log^3 n)$ worst case update time. In *SODA*, pages 470–489, 2017.

[BHNT15] Sayan Bhattacharya, Monika Henzinger, Danupon Nanongkai, and Charalampos Tsourakakis. Space- and time-efficient algorithm for maintaining dense subgraphs on one-pass dynamic streams. In *ACM Symposium on Theory of Computing (STOC)*, pages 173–182, 2015.

[BI04] Daniel Bienstock and Garud Iyengar. Faster approximation algorithms for packing and covering problems. 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.

[BK16] Greg Bodwin and Sebastian Krinninger. Fully dynamic spanners with worst-case update time. In *ESA*, pages 17:1–17:18, 2016.

[BK19] Sayan Bhattacharya and Janardhan Kulkarni. Deterministically maintaining a $(2+\varepsilon)$-approximate minimum vertex cover in $o(1/\varepsilon^2)$ amortized update time. In *SODA*, 2019.

[BK21] Sayan Bhattacharya and Peter Kiss. Deterministic rounding of dynamic fractional matchings. *arXiv preprint arXiv:2105.01615*, 2021.

[BKS12] Surender Baswana, Sumeet Khurana, and Soumojit Sarkar. Fully dynamic randomized algorithms for graph spanners. *ACM Trans. Algorithms*, 8(4):35:1–35:51, 2012. Announced at SODA'08.

[BKT00] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. Data Provenance: Some Basic Issues. *Lecture Notes in Computer Science*, pages 87–93, 2000.

[BKV12] Bahman Bahmani, Ravi Kumar, and Sergei Vassilvitskii. Densest subgraph in streaming and mapreduce. *Proceedings of the VLDB Endowment*, 5(5), 2012.

[BKW23] Maximilian Böther, Otto Kißig, and Christopher Weyand. Efficiently computing directed minimum spanning trees. In *2023 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 86–95. SIAM, 2023.

[BL98] Randal C. Burns and Darrell D. E. Long. In-place reconstruction of delta compressed files. *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing - PODC '98*, pages 267–275, 1998.

[BLL$^+$21] Jan van den Brand, Yin Tat Lee, Yang P. Liu, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. Minimum cost flows, mdps, and l1-regression in nearly linear time for dense instances. In *STOC*, pages 859–869. ACM, 2021.

[BNS$^+$21] Raef Bassily, Kobbi Nissim, Adam D. Smith, Thomas Steinke, Uri Stemmer, and Jonathan R. Ullman. Algorithmic stability for adaptive data analysis. *SIAM J. Comput.*, 50(3), 2021.

[Bod93] Hans L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '93, page 226–234, New York, NY, USA, 1993. Association for Computing Machinery.

[Bod98] Hans L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theoretical Computer Science*, 209(1):1–45, 1998.

[BPW19] Aaron Bernstein, Maximilian Probst, and Christian Wulff-Nilsen. Decremental strongly-connected components and single-source reachability in near-linear time. In *STOC*, pages 365–376, 2019.

[BSS22] Sayan Bhattacharya, Thatchaphol Saranurak, and Pattara Sukprasert. Simple dynamic spanners with near-optimal recourse against an adaptive adversary. In Shiri Chechik, Gonzalo Navarro, Eva Rotenberg, and Grzegorz Herman, editors, *30th Annual European Symposium on Algorithms, ESA 2022, September 5-9, 2022, Berlin/Potsdam, Germany*, volume 244 of *LIPIcs*, pages 17:1–17:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[BT09] Amir Beck and Marc Teboulle. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM journal on imaging sciences*, 2(1):183–202, 2009.

[BV04] Paolo Boldi and Sebastiano Vigna. The webgraph framework I: compression techniques. In *Proceedings of the 13th International Conference on World Wide Web*, pages 595–602, 2004.

[BV14] Ashwinkumar Badanidiyuru and Jan Vondrák. Fast algorithms for maximizing submodular functions. 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 1497–1514. SIAM, 2014.

[BW21] Jackson Brown and Nicholas Weber. DSDB: An Open-Source System for Database Versioning & Curation. *2021 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*, 00:299–307, 2021.

[CD17] Amit Chavan and Amol Deshpande. DEX: Query Execution in a Delta-based Storage System. *Proceedings of the 2017 ACM International Conference on*

Management of Data, pages 171–186, 2017.

[CDE+18] Parinya Chalermsook, Syamantak Das, Guy Even, Bundit Laekhanukit, and Daniel Vaz. Survivable network design for group connectivity in low-treewidth graphs. *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, 2018.

[CGH+05] Julia Chuzhoy, Sudipto Guha, Eran Halperin, Sanjeev Khanna, Guy Kortsarz, Robert Krauthgamer, and Joseph (Seffi) Naor. Asymmetric k-center is log* n-hard to approximate. *J. ACM*, 52(4):538–551, jul 2005.

[CGH+20] Li Chen, Gramoz Goranci, Monika Henzinger, Richard Peng, and Thatchaphol Saranurak. Fast dynamic cuts, distances and effective resistances via vertex sparsifiers. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1135–1146. IEEE, 2020.

[CGL+19] Julia Chuzhoy, Yu Gao, Jason Li, Danupon Nanongkai, Richard Peng, and Thatchaphol Saranurak. A deterministic algorithm for balanced cut with applications to dynamic connectivity, flows, and beyond. *CoRR*, abs/1910.08025, 2019.

[CGSZ04] Artur Czumaj, Michelangelo Grigni, Papa Sissokho, and Hairong Zhao. Approximation schemes for minimum 2-edge-connected and biconnected subgraphs in planar graphs. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 496–505. Society for Industrial and Applied Mathematics, 2004.

[Cha00] Moses Charikar. Greedy approximation algorithms for finding dense components in a graph. In *Approximation Algorithms for Combinatorial Optimization*, pages 84–95, 2000.

[CHK16] Keren Censor-Hillel, Elad Haramaty, and Zohar S. Karnin. Optimal dynamic distributed MIS. In *Proceedings of the 2016 ACM Symposium on Principles* *of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 217–226, 2016.

[CHKZ03] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003.

[CHN$^+$22] Parinya Chalermsook, Chien-Chung Huang, Danupon Nanongkai, Thatchaphol Saranurak, Pattara Sukprasert, and Sorrachai Yingchareonthawornchai. Approximating k-edge-connected spanning subgraphs via a near-linear time LP solver. In Mikolaj Bojanczyk, Emanuela Merelli, and David P. Woodruff, editors, *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022, July 4-8, 2022, Paris, France*, volume 229 of *LIPIcs*, pages 37:1–37:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[CHPQ20] Chandra Chekuri, Sariel Har-Peled, and Kent Quanrud. Fast lp-based approximations for geometric packing and covering problems. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1019–1038. SIAM, 2020.

[Chu21] Julia Chuzhoy. Decremental all-pairs shortest paths in deterministic near-linear time. In *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, pages 626–639, 2021.

[CHv$^+$22] Aleksander B. G. Christiansen, Jacob Holm, Ivor van der Hoog, Eva Rotenberg, and Chris Schwiegelshohn. Adaptive out-orientations with applications. *CoRR*, abs/2209.14087, 2022.

[CK19] Julia Chuzhoy and Sanjeev Khanna. A new algorithm for decremental single-source shortest paths with applications to vertex-capacitated flow and cut problems. In *STOC*, pages 389–400, 2019.

[CKK02] Béla Csaba, Marek Karpinski, and Piotr Krysta. Approximability of dense and sparse instances of minimum 2-connectivity, tsp and path problems. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 74–83. Society for Industrial and Applied Mathematics, 2002.

[CKL+22] 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, 2022.

[CL99] Artur Czumaj and Andrzej Lingas. On approximability of the minimum-cost k-connected spanning subgraph problem. In *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 281–290. Citeseer, 1999.

[CL00] Artur Czumaj and Andrzej Lingas. Fast approximation schemes for euclidean multi-connectivity problems. In *International Colloquium on Automata, Languages, and Programming*, pages 856–868. Springer, 2000.

[CL07] Artur Czumaj and Andrzej Lingas. Approximation schemes for minimum-cost k-connectivity problems in geometric graphs. In *Handbook of Approximation Algorithms and Metaheuristics*. Chapman and Hall/CRC, 2007.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[CQ17a] Chandra Chekuri and Kent Quanrud. Approximating the held-karp bound for metric TSP in nearly-linear time. *CoRR*, abs/1702.04307, 2017.

[CQ17b] Chandra Chekuri and Kent Quanrud. Near-linear time approximation schemes for some implicit fractional packing problems. In *ACM-SIAM Symposium on Discrete Algorithms*, 2017.

[CQ18] Chandra Chekuri and Kent Quanrud. Fast approximations for metric-tsp via linear programming. *CoRR*, abs/1802.01242, 2018.

[CQ22] Chandra Chekuri and Kent Quanrud. $(1-\varepsilon)$-approximate fully dynamic densest subgraph: linear space and faster update time, 2022.

[CQT22] Chandra Chekuri, Kent Quanrud, and Manuel R. Torres. Densest subgraph: Supermodularity, iterative peeling, and flow. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1531–1555, 2022.

[Cre97] P. Crescenzi. A short guide to approximation preserving reductions. In *Proceedings of Computational Complexity. Twelfth Annual IEEE Conference*, pages 262–273, 1997.

[CS15] Markus Chimani and Joachim Spoerhase. Network Design Problems with Bounded Distances via Shallow-Light Steiner Trees. In Ernst W. Mayr and Nicolas Ollinger, editors, *32nd International Symposium on Theoretical Aspects of Computer Science (STACS 2015)*, volume 30 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 238–248, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[CS21] Julia Chuzhoy and Thatchaphol Saranurak. Deterministic algorithms for decremental shortest paths via layered core decomposition. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 2478–2496. SIAM, 2021.

[CZ19] Shiri Chechik and Tianyi Zhang. Fully dynamic maximal independent set in expected poly-log update time. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 370–381, 2019.

[CZ20] Shiri Chechik and Tianyi Zhang. Dynamic low-stretch spanning trees in subpolynomial time. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 463–475. SIAM, 2020.

[DBS17] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 293–304, 2017.

[DBS18] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018.

[DCS17] Maximilien Danisch, T.-H. Hubert Chan, and Mauro Sozio. Large scale density-friendly graph decomposition via convex programming. In *Proceedings of the 26th International Conference on World Wide Web*, page 233–242, 2017.

[DFH$^+$15] Cynthia Dwork, Vitaly Feldman, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Aaron Leon Roth. Preserving statistical validity in adaptive data analysis. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pages 117–126, 2015.

[DHZ22] Riccardo Dondi, Mohammad Mehdi Hosseinzadeh, and Italo Zoppis. Dense temporal subgraphs in protein-protein interaction networks. In *International Conference on Computational Science*, volume 13351, pages 469–480, 2022.

[DJD$^+$09] Xiaoxi Du, Ruoming Jin, Liang Ding, Victor E. Lee, and John H. Thornton. Migration motif: A spatial - temporal pattern mining approach for financial markets. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, page 1135–1144, 2009.

[DLR$^+$22] Laxman Dhulipala, Quanquan C. Liu, Sofya Raskhodnikova, Jessica Shi, Julian Shun, and Shangdi Yu. Differential privacy from locally adjustable graph algorithms: k-core decomposition, low out-degree ordering, and densest subgraphs. In *63rd IEEE Annual Symposium on Foundations of Computer Science*, pages 754–765, 2022.

[Doe20] Benjamin Doerr. *Probabilistic Tools for the Analysis of Randomized Optimization Heuristics*, pages 1–87. Springer International Publishing, Cham, 2020.

[Dol19] Dolt. https://github.com/dolthub/dolt, 2019. last accessed: 13-Oct-22.

[DRMK$^+$22] Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Zoi Kaoudi, Tilmann Rabl, and Volker Markl. Materialization and reuse optimizations for production data science pipelines. SIGMOD '22, page 1962–1976, New York, NY, USA, 2022. Association for Computing Machinery.

[DVC17] DVC. https://github.com/iterative/dvc, 2017. last accessed: 13-Oct-22.

[Elk11] Michael Elkin. Streaming and fully dynamic centralized algorithms for constructing and maintaining sparse spanners. *ACM Trans. Algorithms*, 7(2):20:1–20:17, 2011. Announced at ICALP'07.

[Epp92] David Eppstein. Parallel recognition of series-parallel graphs. *Information and Computation*, 98(1):41–55, 1992.

[Erd86] Paul Erdös. Two problems in extremal graph theory. *Graphs and Combinatorics*, 2(1):189–190, 1986.

[Eul41] Leonhard Euler. Solutio problematis ad geometriam situs pertinentis. *Commentarii academiae scientiarum Petropolitanae*, pages 128–140, 1741.

[FAK$^+$18] Raul Castro Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Sam Madden, and Michael Stonebraker. Aurum: A Data Discovery System. *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1001–1012, 2018.

[Fei98] Uriel Feige. A threshold of ln n for approximating set cover. *J. ACM*, 45(4):634–652, jul 1998.

[Fer98] Cristina G Fernandes. A better approximation ratio for the minimum sizek-edge-connected spanning subgraph problem. *Journal of Algorithms*, 28(1):105–124, 1998.

[FG19] Sebastian Forster and Gramoz Goranci. Dynamic low-stretch trees via dynamic low-diameter decompositions. In *STOC*, pages 377–388. ACM, 2019.

[FGKS18] Samuel Fiorini, Martin Groß, Jochen Könemann, and Laura Sanità. Approximating weighted tree augmentation via chvátal-gomory cuts. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 817–831. SIAM, 2018.

[FHL05] Uriel Feige, MohammadTaghi Hajiaghayi, and James R. Lee. Improved approximation algorithms for minimum-weight vertex separators. In *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '05, page 563–572, New York, NY, USA, 2005. Association for Computing Machinery.

[FHS22] Alireza Farhadi, Mohammad Taghi Hajiaghai, and Elaine Shi. Differentially private densest subgraph. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 151, pages 11581–11597, 2022.

[FJ81] Greg N. Frederickson and Joseph JáJá. Approximation algorithms for several graph augmentation problems. *SIAM J. Comput.*, 10(2):270–283, 1981.

[FKPS16] Andreas Emil Feldmann, Jochen Könemann, Kanstantsin Pashkovich, and Laura Sanità. Fast approximation algorithms for the generalized survivable network design problem. In *ISAAC*, volume 64 of *LIPIcs*, pages 33:1–33:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[Fle00] Lisa Fleischer. Approximating fractional multicommodity flow independent of the number of commodities. *SIAM J. Discrete Math.*, 13(4):505–520, 2000.

[Fle04] Lisa Fleischer. A fast approximation scheme for fractional covering problems with variable upper bounds. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1001–1010, 2004.

[FLS$^+$18] Fedor V. Fomin, Daniel Lokshtanov, Saket Saurabh, MichaŁ Pilipczuk, and Marcin Wrochna. Fully polynomial-time parameterized computations for graphs and matrices of low treewidth. *ACM Trans. Algorithms*, 14(3), jun 2018.

[FNBB06] E. Fratkin, B. T. Naughton, D. L. Brutlag, and S. Batzoglou. Motifcut: regulatory motifs finding with maximum density subgraphs. In *ISMB*, pages 156–157, 2006.

[FTV15] Fedor V. Fomin, Ioan Todinca, and Yngve Villanger. Large induced subgraphs via triangulations and cmso. *SIAM Journal on Computing*, 44(1):54–87, 2015.

[FYC$^+$19] Yixiang Fang, Kaiqiang Yu, Reynold Cheng, Laks V. S. Lakshmanan, and Xuemin Lin. Efficient algorithms for densest subgraph discovery. *Proc. VLDB Endow.*, 12(11):1719–1732, 2019.

[Gab95] Harold N Gabow. A matroid approach to finding edge connectivity and packing arborescences. *Journal of Computer and System Sciences*, 50(2):259–273, 1995.

[Gao12] Yong Gao. Treewidth of erdős–rényi random graphs, random intersection graphs, and scale-free random graphs. *Discrete Applied Mathematics*, 160(4-5):566–578, 2012.

[GGST86] Harold N Gabow, Zvi Galil, Thomas Spencer, and Robert E Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122, 1986.

[GGT89] Giorgio Gallo, Michael D Grigoriadis, and Robert E Tarjan. A fast parametric maximum flow algorithm and applications. *SIAM Journal on Computing*, 18(1):30–55, 1989.

[GGTW09] Harold N Gabow, Michel X Goemans, Éva Tardos, and David P Williamson. Approximating the smallest k-edge connected spanning subgraph by lp-rounding. *Networks: An International Journal*, 53(4):345–357, 2009.

[GGW98] Harold N. Gabow, Michel X. Goemans, and David P. Williamson. An efficient approximation algorithm for the survivable network design problem. *Math. Program.*, 82:13–40, 1998. announced at IPCO'93.

[git05] Git. https://github.com/git/git, 2005. last accessed: 13-Oct-22.

[GJL$^+$13] Aristides Gionis, Flavio Junqueira, Vincent Leroy, Marco Serafini, and Ingmar Weber. Piggybacking on social networks. *Proc. VLDB Endow.*, 6(6):409–420, apr 2013.

[GJN$^+$21] Varun Gupta, Christopher Jung, Seth Neel, Aaron Roth, Saeed Sharifi-Malvajerdi, and Chris Waites. Adaptive machine unlearning. *arXiv preprint arXiv:2106.04378*, 2021.

[GK07] Naveen Garg and Jochen Könemann. Faster and simpler algorithms for multicommodity flow and other fractional packing problems. *SIAM J. Comput.*, 37(2):630–652, 2007.

[GK14] Anupam Gupta and Amit Kumar. Online steiner tree with deletions. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 455–467. SIAM, 2014.

[GKS14] Anupam Gupta, Amit Kumar, and Cliff Stein. Maintaining assignments online: Matching, scheduling, and flows. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 468–479. SIAM, 2014.

[GKZ18] Fabrizio Grandoni, Christos Kalaitzis, and Rico Zenklusen. Improved approximation for tree augmentation: saving by rewiring. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, pages 632–645, 2018.

[GL78] GB Gens and YV Levner. Approximate algorithms for certain universal problems in scheduling theory. *Engineering Cybernetics*, 16(6):31–36, 1978.

[GL94] George Gens and Eugene Levner. A fast approximation algorithm for the subset-sum problem. *INFOR: Information Systems and Operational Research*, 32(3):143–148, 1994.

[GL20] Anupam Gupta and Roie Levin. Fully-dynamic submodular cover with bounded recourse. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1147–1157. IEEE, 2020.

[GLM19] Mohsen Ghaffari, Silvio Lattanzi, and Slobodan Mitrović. Improved parallel algorithms for density-based network clustering. In *Proceedings of the 36th International Conference on Machine Learning*, pages 2201–2210, 2019.

[GLS$^+$23] Anxin (Bob) Guo, Jingwei (Sofia) Li, Pattara Sukprasert, Samir Khuller, Amol Deshpande, and Koyel Mukherjee. To Store or Not to Store: a graph theoretical approach for Dataset Versioning. Under submission, 2023.

[GN22] Rohan Ghuge and Viswanath Nagarajan. Quasi-polynomial algorithms for submodular tree orienteering and directed network design problems. *Mathematics of Operations Research*, 47(2):1612–1630, 2022.

[Gol84] Andrew V Goldberg. Finding a maximum density subgraph. 1984.

[Gon85] Teofilo F Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical computer science*, 38:293–306, 1985.

[GP17] Ofer Grossman and Merav Parter. Improved deterministic distributed construction of spanners. In *DISC*, pages 24:1–24:16, 2017.

[GRST21] Gramoz Goranci, Harald Räcke, Thatchaphol Saranurak, and Zihan Tan. The expander hierarchy and its applications to dynamic graph algorithms. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 2212–2228. SIAM, 2021.

[GSS82] Leslie M. Goldschlager, Ralph A. Shaw, and J. Staples. The maximum flow problem is log space complete for p. *Theoretical Computer Science*, 21:105–111, 1982.

[Gur22] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2022.

[Gut80] Frederick Guthrie. 9. note on the colouring of maps. *Proceedings of the Royal Society of Edinburgh*, 10:727–728, 1880.

[GWN20a] Maximilian Probst Gutenberg and Christian Wulff-Nilsen. Decremental SSSP in weighted digraphs: Faster and against an adaptive adversary. In *SODA*, pages 2542–2561, 2020.

[GWN20b] Maximilian Probst Gutenberg and Christian Wulff-Nilsen. Deterministic algorithms for decremental approximate shortest paths: Faster and simpler. In *SODA*, pages 2522–2541, 2020.

[GWW20] Maximilian Probst Gutenberg, Virginia Vassilevska Williams, and Nicole Wein. New algorithms and hardness for incremental single-source shortest paths in directed graphs. In *Symposium on Theory of Computing*, 2020.

[Haj01] Mohammad Taghi Hajiaghayi. *Algorithms for graphs of (locally) bounded treewidth*. PhD thesis, Citeseer, 2001.

[HDLT01] Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM (JACM)*, 48(4):723–760, 2001.

[HHZ21] Bernhard Haeupler, D. Ellis Hershkowitz, and Goran Zuzic. Tree embeddings for hop-constrained network design. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2021, page 356–369, New York, NY, USA, 2021. Association for Computing Machinery.

[HKM$^+$20] Avinatan Hasidim, Haim Kaplan, Yishay Mansour, Yossi Matias, and Uri Stemmer. Adversarially robust streaming algorithms via differential privacy. *Advances in Neural Information Processing Systems*, 33, 2020.

[HKS09] Mohammad Taghi Hajiaghayi, Guy Kortsarz, and Mohammad R Salavatipour. Approximating buy-at-bulk and shallow-light k-steiner trees. *Algorithmica*, 53(1):89–103, 2009.

[HQC22] Elfarouk Harb, Kent Quanrud, and Chandra Chekuri. Faster and scalable algorithms for densest subgraph and decomposition. In *Advances in Neural Information Processing Systems*, 2022.

[HR20a] Jacob Holm and Eva Rotenberg. Fully-dynamic planarity testing in polylogarithmic time. In *STOC*, pages 167–180. ACM, 2020.

[HR20b] Jacob Holm and Eva Rotenberg. Worst-case polylog incremental spqr-trees: Embeddings, planarity, and triconnectivity. In *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 2378–2397, 2020.

[HSB$^+$16] Bryan Hooi, Hyun Ah Song, Alex Beutel, Neil Shah, Kijung Shin, and Christos Faloutsos. FRAUDAR: Bounding graph fraud in the face of camouflage. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, page 895–904, 2016.

[HSSC08] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[HU14] Moritz Hardt and Jonathan Ullman. Preventing false discovery in interactive data analysis is hard. In *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, pages 454–463. IEEE, 2014.

[HVT98] James J. Hunt, Kiem-Phong Vo, and Walter F. Tichy. Delta algorithms: an empirical analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7(2):192–214, 1998.

[HXL$^+$20] Silu Huang, Liqi Xu, Jialin Liu, Aaron J. Elmore, and Aditya Parameswaran. ORPHEUSDB: bolt-on versioning for relational databases (extended version). *The VLDB Journal*, 29(1):509–538, 2020.

[IK75] Oscar H. Ibarra and Chul E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *J. ACM*, 22(4):463–468, oct 1975.

[Jai01] Kamal Jain. A factor 2 approximation algorithm for the generalized steiner network problem. *Combinatorica*, 21(1):39–60, 2001.

[Jaj92] J. Jaja. *Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992.

[Jan01] Klaus Jansen. Approximation algorithms for fractional covering and packing problems, and applications. In *International Symposium on Fundamentals of Computation Theory*, 2001.

[JJ19] Yasith Jayawardana and Sampath Jayarathna. DFS: A Dataset File System for Data Discovering Users. *2019 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*, 00:355–356, 2019.

[JMS02] Kamal Jain, Mohammad Mahdian, and Amin Saberi. A new greedy approach for facility location problems. In *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*, STOC '02, page 731–740, New York, NY, USA, 2002. Association for Computing Machinery.

[JRST01] Thor Johnson, Neil Robertson, P. D. Seymour, and Robin Thomas. Directed tree-width. *Journal of Combinatorial Theory. Series B*, 82(1):138–154, May 2001. Funding Information: 1Partially supported by the NSF under Grant DMS-9701598. 2 Research partially supported by the DIMACS Center, Rutgers University, New Brunswick, NJ 08903. 3Partially supported by the NSF under Grant DMS-9401981. 4Partially supported by the ONR under Contact N00014-97-1-0512. 5Partially supported by the NSF under Grant DMS-9623031 and by the NSA under Contract MDA904-98-1-0517.

[JXRF09] Ruoming Jin, Yang Xiang, Ning Ruan, and David Fuhry. 3-HOP: A high-compression indexing scheme for reachability query. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, page 813–826, 2009.

[Kar75] Richard M Karp. The fast approximate solution of hard combinatorial problems. In *Proc. 6th South-Eastern Conf. Combinatorics, Graph Theory and Computing (Florida Atlantic U. 1975)*, pages 15–31, 1975.

[Kar99] David R. Karger. Random sampling in cut, flow, and network design problems. *Math. Oper. Res.*, 24(2):383–413, 1999.

[Kar00] David R Karger. Minimum cuts in near-linear time. *Journal of the ACM (JACM)*, 47(1):46–76, 2000. announced at STOC'96.

[KD12] Udayan Khurana and Amol Deshpande. Efficient Snapshot Retrieval over Historical Graph Data. *arXiv*, 2012. Graph database systems — stroing dynamic graphs so that a graph at a specific time can be queried. Vertices are marked with bits encoding information on which versions it belong to.

[Kis21] Peter Kiss. Deterministic dynamic matching in worst-case update time. *arXiv preprint arXiv:2108.10461*, 2021.

[KKOV07] Rohit Khandekar, Subhash Khot, Lorenzo Orecchia, and Nisheeth K Vishnoi. On a cut-matching game for the sparsest cut problem. *Univ. California, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2007-177*, 2007.

[KKT95] David R Karger, Philip N Klein, and Robert E Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM (JACM)*, 42(2):321–328, 1995.

[KMNS21] Haim Kaplan, Yishay Mansour, Kobbi Nissim, and Uri Stemmer. Separating adaptive streaming from oblivious streaming. *arXiv preprint arXiv:2101.10836*, 2021.

[KMPS03] Hans Kellerer, Renata Mansini, Ulrich Pferschy, and Maria Grazia Speranza. An efficient fully polynomial approximation scheme for the subset-sum problem. *Journal of Computer and System Sciences*, 66(2):349–370, 2003.

[Kor22] Tuukka Korhonen. A single-exponential time 2-approximation algorithm for treewidth. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 184–192, 2022.

[KP97] Guy Kortsarz and David Peleg. Approximating shallow-light trees. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 103–110, 1997.

[KRRT99] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew Tomkins. Trawling the web for emerging cyber-communities. *Computer Networks*, 31(11):1481–1493, 1999.

[KRV09] Rohit Khandekar, Satish Rao, and Umesh V. Vazirani. Graph partitioning using single commodity flows. *J. ACM*, 56(4):19:1–19:15, 2009.

[KRY95] Samir Khuller, Balaji Raghavachari, and Neal E. Young. Balancing minimum spanning trees and shortest-path trees. *Algorithmica*, 14(4):305–321, 1995.

[KS09] Samir Khuller and Barna Saha. On finding dense subgraphs. In Susanne Albers, Alberto Marchetti-Spaccamela, Yossi Matias, Sotiris E. Nikoletseas, and Wolfgang Thomas, editors, *International Colloquium on Automata, Languages and Programming*, volume 5555, pages 597–608, 2009.

[KS16] M. Reza Khani and Mohammad R. Salavatipour. Improved approximations for buy-at-bulk and shallow-light k-steiner trees and (k,2)-subgraph. *J. Comb. Optim.*, 31(2):669–685, feb 2016.

[KT93] Samir Khuller and Ramakrishna Thurimella. Approximation algorithms for graph augmentation. *Journal of Algorithms*, 14(2):214–225, 1993.

[KV94] Samir Khuller and Uzi Vishkin. Biconnectivity approximations and graph carvings. *J. ACM*, 41(2):214–235, 1994. announced at STOC'92.

[KY99] Philip N. Klein and N. Young. On the number of iterations for dantzig-wolfe optimization and packing-covering approximation algorithms. *SIAM J. Comput.*, 44:1154–1172, 1999.

[Lak20] LakeFS. https://github.com/treeverse/lakeFS, 2020. last accessed: 13-Oct-22.

[LGS12] Bundit Laekhanukit, Shayan Oveis Gharan, and Mohit Singh. A rounding by sampling approach to the minimum size k-arc connected subgraph problem. In *International Colloquium on Automata, Languages, and Programming*, pages 606–616. Springer, 2012.

[Li21] Jason Li. Deterministic mincut in almost-linear time. In *STOC*, pages 384–395. ACM, 2021.

[LK14] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. June 2014.

[LNP$^+$21] Jason Li, Danupon Nanongkai, Debmalya Panigrahi, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. Vertex connectivity in polylogarithmic max-flows. In *STOC*, pages 317–329. ACM, 2021.

[LPS21] Jason Li, Debmalya Panigrahi, and Thatchaphol Saranurak. A nearly optimal all-pairs min-cuts algorithm in simple graphs. *arXiv preprint arXiv:2106.02233*, 2021.

[LS21] Jason Li and Thatchaphol Saranurak. Deterministic weighted expander decomposition in almost-linear time. *arXiv preprint arXiv:2106.01567*, 2021.

[LSY$^+$22] Quanquan C. Liu, Jessica Shi, Shangdi Yu, Laxman Dhulipala, and Julian Shun. Parallel batch-dynamic algorithms for k-core decomposition and related graph problems. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 191–204, 2022.

[Mac00] Josh MacDonald. *File system support for delta compression.* PhD thesis, Masters thesis. Department of Electrical Engineering and Computer Science, University of California at Berkley, 2000.

[Mad10a] Aleksander Madry. Fast approximation algorithms for cut-based problems in undirected graphs. In *FOCS*, pages 245–254. IEEE Computer Society, 2010.

[Mad10b] Aleksander Madry. Faster approximation schemes for fractional multicommodity flow problems via dynamic graph algorithms. In *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*, pages 121–130, 2010.

[MGE$^+$16] Michael Maddox, David Goehring, Aaron J. Elmore, Samuel Madden, Aditya Parameswaran, and Amol Deshpande. Decibel: The Relational Dataset Branching System. *Proceedings of the VLDB Endowment. International Conference on Very Large Data Bases*, 9(9):624–635, 2016.

[MRS$^+$98] Madhav V Marathe, Ramamoorthi Ravi, Ravi Sundaram, SS Ravi, Daniel J Rosenkrantz, and Harry B Hunt III. Bicriteria network design problems. *Journal of algorithms*, 28(1):142–171, 1998.

[MSM$^+$22] Naga Nithin Manne, Shilvi Satpati, Tanu Malik, Amitabha Bagchi, Ashish Gehani, and Amitabh Chaudhary. CHEX: multiversion replay with ordered checkpoints. *Proceedings of the VLDB Endowment*, 15(6):1297–1310, 2022.

[MSS$^+$23] Koyel Mukherjee, Raunak Shah, Shiv K. Saini, Karanpreet Singh, Khushi , Harsh Kesarwani, Kavya Barnwal, and Ayush Chauhan. Towards optimizing storage costs on the cloud. *IEEE 39th International Conference on Data Engineering (ICDE) (To Appear)*, 2023.

[MVLB15] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. The graph structure in the web - analyzed on different aggregation levels. *J. Web Sci.*, 1:33–47, 2015.

[Nag06] William Nagel. Subversion: not just for code anymore. *Linux Journal*, 2006(143):10, 2006.

[Nes83] Yurii Evgen'evich Nesterov. A method of solving a convex programming problem with convergence rate $o\bigl(\frac{1}{k^2}\bigr)$. In *Doklady Akademii Nauk*, volume 269, pages 543–547. Russian Academy of Sciences, 1983.

[NS17a] Danupon Nanongkai and Thatchaphol Saranurak. Dynamic spanning forest with worst-case update time: adaptive, las vegas, and o (n1/2-ε)-time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 1122–1129, 2017.

[NS17b] Danupon Nanongkai and Thatchaphol Saranurak. Dynamic spanning forest with worst-case update time: adaptive, Las vegas, and $O(n^{1/2-\varepsilon})$-time. In *STOC*, pages 1122–1129, 2017.

[NSW17] Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. Dynamic minimum spanning forest with subpolynomial worst-case update time. In *FOCS*, pages 950–961. IEEE Computer Society, 2017.

[NV21] Dung Nguyen and Anil Vullikanti. Differentially private densest subgraph detection. In *Proceedings of the 38th International Conference on Machine Learning*, pages 8140–8151, 2021.

[NZM$^+$19] Fatemeh Nargesian, Erkang Zhu, Renée J. Miller, Ken Q. Pu, and Patricia C. Arocena. Data lake management: Challenges and opportunities. *Proc. VLDB Endow.*, 12(12):1986–1989, aug 2019.

[pac16] Pachyderm. https://github.com/pachyderm/pachyderm, 2016. last accessed: 13-Oct-22.

[Pri10] David Pritchard. k-edge-connectivity: Approximation and LP relaxation. In *WAOA*, volume 6534 of *Lecture Notes in Computer Science*, pages 225–236. Springer, 2010.

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

[PST91] Serge A. Plotkin, David B. Shmoys, and Éva Tardos. Fast approximation algorithms for fractional packing and covering problems. *[1991] Proceedings 32nd Annual Symposium of Foundations of Computer Science*, pages 495–504, 1991.

[RA15] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[Ram90] Rajeev Raman. The power of collision: Randomized parallel algorithms for chaining and integer sorting. In *Foundations of Software Technology and Theoretical Computer Science*, pages 161–175, 1990.

[Rav94] R. Ravi. Rapid rumor ramification: approximating the minimum broadcast time. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 202–213, 1994.

[RFT22] Paul Roome, Tao Feng, and Sachin Thakur. Announcing the availability of data lineage with unity catalog. https://www.databricks.com/blog/2022/06/08/announcing-the-availability-of-data-lineage-with-unity-catalog.html, 2022. last accessed: 13-Oct-22.

[RTG14] Polina Rozenshtein, Nikolaj Tatti, and Aristides Gionis. Discovering dynamic communities in interaction networks. In *Machine Learning and Knowledge Discovery in Databases*, pages 678–693, 2014.

[Sch03] Alexander Schrijver. *Combinatorial optimization: polyhedra and efficiency*, volume 24. Springer Science & Business Media, 2003.

[SCMMS12] Adam Seering, Philippe Cudre-Mauroux, Samuel Madden, and Michael Stonebraker. Efficient Versioning for Scientific Array Databases. *2012 IEEE 28th International Conference on Data Engineering*, 1:1013–1024, 2012.

[SGT$^+$08] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

[She09] Jonah Sherman. Breaking the multicommodity flow barrier for o(vlog n)-approximations to sparsest cut. In *FOCS*, pages 363–372. IEEE Computer Society, 2009.

[She13] Jonah Sherman. Nearly maximum flows in nearly linear time. In *FOCS*, pages 263–269. IEEE Computer Society, 2013.

[SHK$^+$10] Barna Saha, Allison Hoch, Samir Khuller, Louiqa Raschid, and Xiao-Ning Zhang. Dense subgraphs with restrictions and applications to gene annotation graphs. In *Annual International Conference on Research in Computational Molecular Biology*, 2010.

[SKS$^+$19] Maximilian E Schule, Lukas Karnowski, Josef Schmeißer, Benedikt Kleiner, Alfons Kemper, and Thomas Neumann. Versioning in Main-Memory Database

Systems: From MusaeusDB to TardisDB. *Proceedings of the 31st International Conference on Scientific and Statistical Database Management*, pages 169–180, 2019.

[SLDS23] Pattara Sukprasert, Quanquan Liu, Laxman Dhulipala, and Julian Shun. Practical Parallel Algorithms for Near-Optimal Densest Subgraphs on Massive Graphs. Under submission, 2023.

[SO06] Roberto Solis-Oba. *Approximation Algorithms for the k-Median Problem*, pages 292–320. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[SPG$^+$] Yogesh L Simmhan, Beth Plale, Dennis Gannon, et al. A survey of data provenance techniques.

[SSA$^+$15] Tripti Swarnkar, Sérgio Nery Simões, Anji Anurak, Helena Brentani, Jyotirmoy Chatterjee, Ronaldo Fumio Hashimoto, David Correa Martins, and Pabitra Mitra. Identifying dense subgraphs in protein-protein interaction network for gene selection from microarray data. *Netw. Model. Anal. Health Informatics Bioinform.*, 4(1):33, 2015.

[SU17] Thomas Steinke and Jonathan Ullman. Tight lower bounds for differentially private selection. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 552–563. IEEE, 2017.

[Sue02] Dimitre Trendafilov Nasir Memon Torsten Suel. zdelta: An efficient delta compression tool. 2002.

[SV20] Hsin-Hao Su and Hoa T. Vu. Distributed Dense Subgraph Detection and Low Outdegree Orientation. In *34th International Symposium on Distributed Computing*, pages 15:1–15:18, 2020.

[SW19] Thatchaphol Saranurak and Di Wang. Expander decomposition and pruning: Faster, stronger, and simpler. In *SODA*, pages 2616–2635. SIAM, 2019.

[SW20] Shay Solomon and Nicole Wein. Improved dynamic graph coloring. *ACM Trans. Algorithms*, 16(3), June 2020.

[Ter19] TerminusDB. https://github.com/terminusdb/terminusdb, 2019. last accessed: 13-Oct-22.

[TG15] Nikolaj Tatti and Aristides Gionis. Density-friendly graph decomposition. In *Proceedings of the 24th International Conference on World Wide Web*, page 1089–1099, 2015.

[Tho05] Mikkel Thorup. Worst-case update times for fully-dynamic all-pairs shortest paths. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 112–119, 2005.

[vdBLN$^+$20] Jan van den Brand, Yin Tat Lee, Danupon Nanongkai, Richard Peng, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. Bipartite matching in nearly-linear time on moderately dense graphs. In *FOCS*, pages 919–930. IEEE, 2020.

[Vis10] Uzi Vishkin. Thinking in parallel: Some basic data-parallel algorithms and techniques. *Parallel Algorithms*, 2010.

[Waj20] David Wajc. Rounding dynamic matchings against an adaptive adversary. *Symposium on Theory of Computing*, 2020.

[WDL$^+$18] Sheng Wang, Tien Tuan Anh Dinh, Qian Lin, Zhongle Xie, Meihui Zhang, Qingchao Cai, Gang Chen, Beng Chin Ooi, and Pingcheng Ruan. Forkbase: An efficient storage engine for blockchain and forkable applications. *Proc. VLDB Endow.*, 11(10):1137–1150, jun 2018.

[Wen91] Rephael Wenger. Extremal graphs with no c4's, c6's, or c10's. *Journal of Combinatorial Theory, Series B*, 52(1):113–116, 1991.

[WPC$^+$20] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 32(1):4–24, 2020.

[Wul17] Christian Wulff-Nilsen. Fully-dynamic minimum spanning forest with improved worst-case update time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 1130–1143, 2017.

[WZ20] David P Woodruff and Samson Zhou. Tight bounds for adversarially robust streams and sliding windows via difference estimators. *arXiv preprint arXiv:2011.07471*, 2020.

[XJF$^+$14] Wen Xia, Hong Jiang, Dan Feng, Lei Tian, Min Fu, and Yukun Zhou. Ddelta: A deduplication-inspired fast delta compression approach. *Performance Evaluation*, 79:258–272, 2014.

[YCJ20] Tangwei Ying, Hanhua Chen, and Hai Jin. Pensieve: Skewness-aware version switching for efficient graph processing. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 699–713, New York, NY, USA, 2020. Association for Computing Machinery.

[You14] Neal E. Young. Nearly linear-time approximation schemes for mixed packing/-covering and facility-location linear programs. *CoRR*, abs/1407.3015, 2014.

[ZCH$^+$20] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI open*, 1:57–81, 2020.

[Zen14] Rico Zenklusen. Connectivity interdiction. *Oper. Res. Lett.*, 42(6-7):450–454, 2014.

[Zha21] Tianyi Zhang. Faster cut-equivalent trees in simple graphs. *arXiv preprint arXiv:2106.03305*, 2021.

[ZLJG18] Yin Zhang, Huiping Liu, Cheqing Jin, and Ye Guo. Storage and recreation trade-off for multi-version data management. In Yi Cai, Yoshiharu Ishikawa, and Jianliang Xu, editors, *Web and Big Data*, pages 394–409, Cham, 2018. Springer International Publishing.

APPENDIX A

k-Edge Connected Spanning Subgraphs

A.1. Sparsify the fractional solution (Proof of Lemma 2.31)

It suffices to prove the following lemma.

Lemma A.1. *Given a feasible solution x to kECSS, and a non-negative cost function* $: E \to \mathbb{R}_{\geq 0}$*, and $\varepsilon > 0$, there is an algorithm that runs in $\tilde{O}(m)$ time, and w.h.p., outputs another feasible solution y to kECSS such that*

- $\sum_{e \in E} c_e y_e \leq (1+\varepsilon) \sum_{e \in E} c_e x_e$.
- $\text{support}(y) \subseteq \text{support}(x)$.
- $|\text{support}(y)| = O\left(\frac{kn \log n}{\varepsilon^2}\right)$.

We devote the rest of this subsection to proving Lemma A.1.

Let x be a near-optimal kECSS fractional solution obtained by Theorem 2.1. Compute the solution y using Lemma A.1. Create a graph G' by keeping only edges in the support of y.

Before proving the lemma, we first develop an extension to the sparsification theorem from the paper of Benczur and Karger.

We follow the definitions by [BK15, CQ18].

Definition A.2 (Edge stength)**.** *Let $G = (V, E, w)$ be a weighted undirected graph.*

- *G is k-connected if every cut in G has weight at least k.*
- *A k-strong component is a maximal non-empty k-connected vertex-induced subgraph of G.*

- *The strength of an edge e, denoted as κ_e is the maximum k such that both endpoints of e belong to some k-strong component.*

Lemma A.3 ([BK15])**.**

$$\sum_{e \in E} \frac{w_e}{\kappa_e} \leq n - 1$$

Lemma A.4 ([BK15])**.** *In $\tilde{O}(m)$ time, we can compute approximate stength $\tilde{\kappa}_e$ for each edge $e \in E$ such that $\tilde{\kappa}_e \leq \kappa_e$ and $\sum_{e \in E} \frac{w_e}{\tilde{\kappa}_e} = O(n)$*

Given a cut C and a subset $S \subseteq C$ of its edges, where $|S| \leq k - 1$, we say $C\mathcal{S}$ is a *constrained cut*. The next theorem states that all constrained cuts would have their weights closed to their original weights after the sampling.

Theorem A.5 (Extension to Compression Theorem [BK15])**.** *Given $G = (V, E, w)$, let $p : E \to [0, 1]$ be a probability function over edges of G. We construct a random weighted graph $H = (V, E_H, w')$ as follows. For each edge $e \in E$, we independently add edge e into E_H with weight $w'_e = w_e/p_e$, with probability p_e. For $\delta \geq \Omega(kd \log n)$, if $p_e \geq \min\{1, \delta \frac{w_e}{\kappa_e}\}$ for all $e \in E$, then with high probability $\left(\text{over } 1 - \frac{1}{n^d}\right)$, every constrained cut in H has weight between $(1 - \varepsilon)$ and $(1 + \varepsilon)$ times its value in G.*

Proof. This theorem follows almost closely the proof of Benczur-Karger. We sketch here the part where we need a minor modification.

The proof of Benczur-Karger roughly has two components. The first reduces the analysis for general case to the "weighted sum" of the "uniform" cases where the minimum cut is large, i.e. edge weights are at most 1 and minimum cut at least $D = \Omega(kd \log n)$. This first component works exactly the same in our case.

Now in each uniform instance which is the second component of Benczur-Karger, the probabilistic arguments can be made in the following way: For each cut C, since edges are sampled independently, we can use Chernoff bound to upper bound the probability that each cut C deviates more than $(1+\varepsilon)$ factor (after sampling). Let μ_C denote this probability. Therefore, the bad event that there is a cut deviating too much is upper bounded by $\sum_C \mu_C$.

Benczur-Karger analyzes this probability by constructing an auxiliary experiment: Imagine each edge is deleted with probability p, then the sum is exactly the expected number of "empty cuts" in the resulting graph. They upper bound this by using the term $\mathbb{E}[2^R]$ where R is the (random) number of connected components in the resulting graph. They show (using a coupling argument) that $\mathbb{E}[2^R] = O\left(n^2 p^D\right)$, which vanishes whenever $D = \Omega\left(d \log n\right)$. Here is where we need to slightly change the proof. The bad even that we need to bound is not just all the cuts $\left(\sum\limits_C \mu_C\right)$, but also all the constraint cuts. Let $\mu_{C\setminus S}$ be the probability of the bad event that the constraint cut $C \setminus S$ is deviating too much. We want to bound

$$\sum_C \sum_{S \subseteq C, |S| \leq k-1} \mu_{C\setminus S}.$$

We will create, by enumerating, $\binom{m}{k}$ different graphs H so that each H has at most k edges removed from G. Note that all constrained cuts are now defined in these graphs H. In the original sampling, if an edge G is removed, then we remove it similarly in all graphs H (ignoring it is present in H or not).

Given that there are R connected components in H, there are $O(2^R)$ empty cuts. We consider $\binom{m}{k}$ different graphs derived from H by exhaustively remove a subset $S \subseteq E$ of k edges. Some edges in S might already be removed in H, so some configurations will be identical. We now count the empty cuts in these $\binom{m}{k}$ graphs. To upper bound $\sum_C \sum_{S \subseteq C, |S| \leq k-1} \mu_{C\setminus S}$, we just need to compute the total number of "empty cuts" in all these graphs H.

In each H, there are at most $R+k$ connected components. Hence, each graph has at most 2^{R+k} empty cuts. Sum up this number among all the graphs, we get that

$$\sum_C \sum_{S \subseteq C, |S| \leq k-1} \mu_{C\setminus S} \leq \mathbb{E}\left[\binom{m}{k} 2^{R+k}\right].$$

Since $\mathbb{E}[2^R] = O(n^2p^D)$, we get that

$$\sum_{C} \sum_{S\subseteq C, |S|\leq k} \mu_{C\setminus S} = O\left(\binom{m}{k} 2^k n^2 p^D\right) = O\left(\left(\frac{2em}{k}\right)^k n^2 p^D\right),$$

which again vanishes if D is large enough (at least $\Omega(kd\log n)$).

□

We are now ready to prove Lemma A.1. In fact the same proof in [CQ18] can be applied once we have Theorem A.5.

Proof of Lemma A.1. We first use Lemma A.4 to compute approximate edge strength $\tilde{\kappa}_e$ for each edge $e \in E$ so that $\tilde{\kappa}_e \leq \kappa$ and $\sum_{e\in E} \frac{w_e}{\tilde{\kappa}_e} = O(n)$ in $\tilde{O}(m)$ time. Let $\delta = \Theta(kd\log n)$ for some large constant d. Let $\text{cost}(x) = \sum_{e\in E} c_e x_e$. For each edge $e \in E$ let $p_e = \min\{1, \frac{\delta x_e}{\varepsilon^2\tilde{\kappa}_e}\}$, and $q_e = \min\{1, \frac{\delta c_e x_e}{\varepsilon^2 \text{cost}(x)}\}$, and define $r_e = \max(p_e, q_e)$.

We will focus on x from the perspective of kECSS LP with knapsack constraints.

We construct a random graph $H = (V, E', x')$ using r as a probability function over edges of G and we x as weight function of the graph as follows. For each edge $e \in E$, we independently sample edge e into E' with weight $x'_e = x_e/r_e$ with probability r_e. Since

$$r_e = \max(p_e, q_e) \geq p_e = \min\{1, \frac{\delta x_e}{\varepsilon^2 \tilde{\kappa}_e}\} \geq \min\{1, \delta\frac{x_e}{\kappa_e}\}$$

for sufficiently large constant d, by Theorem A.5, we get w.h.p.,

$$\forall C \in \mathcal{C} \forall S \in C, |S| \leq k-1, \sum_{e\in C\setminus S} x'_e \geq (1-\varepsilon) \sum_{e\in C\setminus S} x_e \geq (1-\varepsilon)(k - |S|).$$

Observe that

$$\sum_{e\in E} r_e \leq \sum_{e\in E} p_e + \sum_{e\in E} q_e = O(\frac{n\delta}{\varepsilon^2} + \frac{\delta}{\varepsilon^2}) = O(\frac{n\delta}{\varepsilon^2})$$

By Chernoff bound, we have

$$P(\sum_{e\in E} c_e x'_e \geq (1+\varepsilon)\sum_{e\in E} c_e x_e) \leq \exp(-\Omega(\delta))$$

and,

$$P(|E'| \geq (1+\varepsilon)O(\frac{n\delta}{\varepsilon^2})) \leq \exp(-\delta/\varepsilon^2)$$

By the union bound, we have the followings w.h.p.

$$\sum_{e\in C\setminus S} x'_e \geq (1-\varepsilon)(k-|S|), \quad \forall C \in \mathcal{C}, \forall S \in C, |S| \leq k-1,$$

$$|\operatorname{support}(x')| \leq O(\frac{n\delta}{\varepsilon^2}) \quad \text{and} \sum_{e\in E} c_e x'_e \leq (1+\varepsilon)\sum_{e\in E} c_e x_e$$

Therefore, $y' = (1+\varepsilon)x'$ is a feasible solution to kECSS. Also, $|\operatorname{support}(y')| \leq O(\frac{n\delta}{\varepsilon^2})$, and $\sum_{e\in E} c_e y'_e \leq (1+\varepsilon)^2 \sum_{e\in E} c_e x_e$. Finally, we can get $(1+\varepsilon')\sum_{e\in E} c_e x_e$ by a proper scaling factor for ε.

□

A.2. Bounding the minimum normalized free cut (Proof of Theorem 2.7)

Let us assume that the costs c_e are integers (but they can be exponentially large in values). Karger's sampling [Kar00] gives a near-linear time algorithm to create a skeleton graph H so that all cuts in H are approximately preserved, and the minimum cut value is $O(\log|E|)$. It only requires an easy modification of Karger's arguments to show that we can create a skeleton H such that all k-free minimum cuts are approximately preserved, and that the value of the minimum k-free cuts is $\Theta(k\log|E|)$. We will run our static algorithm in graph H instead. As outlined in Karger's paper [Kar99], the assumption that we do not know the value of the optimal can be resolved by enumerating them in the geometric scales, and the sampling will guarantee that the running time would not blow up by more than a constant factor.

A.3. Multiplicative weight update guarantee (Proof of Theorem 2.16)

The proof is done via duality. The primal and dual solutions will be maintained and updated, until the point where one can argue that their values converge to each other; this implies that both the primal and dual solutions are approximately optimal. Recall the primal LP is the covering LP:

$$\min\{c^T x : Ax \geq 1, x \geq 0\}$$

The dual LP is the following packing LP:

$$\max\{y^T \mathbf{1} : y^T A \leq c^T, y \geq 0\}$$

For the primal LP, we maintain vectors $\mathbf{w}^{(t)} \in \mathbb{R}^n$, where $\mathbf{w}_i^{(0)} = 1/c_i$ for each $i \in [n]$. The tentative primal solution on day t is $\bar{\mathbf{w}}^{(t)} = \mathbf{w}^{(t)}/\textsc{MinRow}(A, \mathbf{w}^{(t)})$. For the dual packing LP, we maintain vectors $\mathbf{f}^{(t)} \in \mathbb{R}^m$ where $\mathbf{f}^{(0)} = \mathbf{0}$. The tentative dual solution on day t is defined as $\bar{\mathbf{f}}^{(t)} = f^{(t)}/\mathsf{cong}(\mathbf{f}^{(t)})$, where $\mathsf{cong}(\mathbf{f})$ is the maximum ratio of violated constraints by $\mathbf{f}$, that is,

$$\mathsf{cong}(\mathbf{f}) = \max_{i \in [n]} \frac{(\mathbf{f}^T A)_i}{c_i}. \tag{A.1}$$

Notice that $\bar{\mathbf{f}}^{(t)}$ is a feasible dual solution on each day.

Now we explain the update rules on each day. Let $j(t)$ be the row that achieves $A_{j(t)}\mathbf{w}^{(t-1)} \leq (1+\varepsilon)\textsc{MinRow}(A, \mathbf{w}^{(t-1)})$.

- Update $\mathbf{f}_{j(t)}^{(t)} \leftarrow \mathbf{f}_{j(t)}^{(t-1)} + \delta(t)$ where $\delta(t) = \min_{i \in [n]} \frac{c_i}{A_{j(t),i}}$ is the "increment" on day t.
- Update $\mathbf{w}_i^{(t)} \leftarrow \mathbf{w}_i^{(t-1)} \exp\left(\varepsilon \cdot \frac{\delta(t) A_{j(t),i}}{c_i}\right)$ for each $i \in [n]$.

Denote the primal value at time t by $P(t) = c^T \bar{\mathbf{w}}^{(t)}$ and the dual by $D(t) = ||\bar{\mathbf{f}}^{(t)}||_1$; so we have $P(t) \geq D(t)$ for all t.

Theorem A.6. *Let t^* be the day t for which $P(t)$ is minimized and $N = \Omega(\frac{n}{\varepsilon^2}\ln n)$ be the total number of days. Then we have that $P(t^*) \le (1 + O(\varepsilon))D(N)$. In particular, $\bar{\boldsymbol{w}}^{(t^*)}$ and $\bar{\boldsymbol{f}}^{(N)}$ are near-optimal primal and dual solutions.*

Our proof relies on the estimates of a potential function defined as $\Phi^{(t)} = c^T\mathbf{w}^{(t)} = \sum_{i\in[n]} c_i\mathbf{w}_i^{(t)}$.

Lemma A.7. *We have, on each day t,*

$$\exp(\varepsilon \cdot \mathsf{cong}(\boldsymbol{f}^{(t)})) \le \Phi^{(t)} \le n \cdot \exp\left(\varepsilon(1+3\varepsilon)\sum_{0<t'\le t}\frac{\delta(t')}{P(t'-1)}\right).$$

Proof. First we show the lower bound of $\Phi^{(t)}$. Fix column $i \in [n]$ such that $\frac{((\mathbf{f}^{(t)})^T A)_i}{c_i} = \mathsf{cong}(\mathbf{f}^{(t)})$. Notice that the value of $c_i\mathbf{w}_i^{(t)}$ is equal to:

$$\exp\left(\frac{\varepsilon}{c_i}\cdot\sum_{t'\le t}\delta(t')A_{j(t'),i}\right).$$

The term $\delta(t')A_{j(t),i}$ is exactly the increase in $((\mathbf{f}^{(t)})^T A)_i$ at time t, so we have that

$$c_i\mathbf{w}_i^{(t)} \ge \exp\left(\frac{\varepsilon}{c_i}\cdot((\mathbf{f}^{(t)})^T A)_i\right) = \exp(\varepsilon\cdot\mathsf{cong}(\mathbf{f}^{(t)})),$$

as desired.

Next, we prove the upper bound on the potential function. Observe that[1] $\mathbf{w}_i^{(t)} \le \mathbf{w}_i^{(t-1)}(1+\varepsilon(1+\varepsilon)\cdot\frac{\delta(t)A_{j(t),i}}{c_i})$. This formula shows the increase of potential at time t to be at most

$$\Phi^{(t)} \le \Phi^{(t-1)} + \sum_{i\in[n]}\varepsilon(1+\varepsilon)\cdot\delta(t)A_{j(t),i}\mathbf{w}_i^{(t-1)} \le \Phi^{(t-1)}\exp\left(\frac{\varepsilon(1+\varepsilon)\delta(t)}{\Phi^{(t-1)}}\cdot\sum_{i\in[n]}A_{j(t),i}\mathbf{w}_i^{(t-1)}\right)$$

[1]In particular, we use the inequality $e^\gamma \le 1+\gamma+\gamma^2$ for $\gamma\in[0,1)$ and the fact that the ratio $\delta(t)A_{j(t),i}/c_i$ is at most 1.

Notice that $\sum_{i\in[n]} A_{j(t),i}\mathbf{w}_i^{(t-1)} = (A_{j(t)}\mathbf{w}^{(t-1)})$ is at most $(1+\varepsilon)\textsc{MinRow}(A,\mathbf{w}^{(t-1)})$ by the choice of the update rules. The term reduces further to:

$$\Phi^{(t)} \leq \Phi^{(t-1)}\exp\left(\frac{\varepsilon(1+\varepsilon)^2\delta(t)}{P(t-1)}\right) \leq \Phi^{(t-1)}\exp\left(\frac{\varepsilon(1+3\varepsilon)\delta(t)}{P(t-1)}.\right)$$

By applying the fact that $\Phi^{(0)} = n$ and the above fact iteratively, we get the desired bound.

□

Finally, we argue that the lemma implies Theorem A.6. Consider the last day N. Taking logarithms on both sides gives us:

$$\mathsf{cong}(\mathbf{f}^{(N)}) \leq \frac{\ln n}{\varepsilon} + (1+3\varepsilon)\sum_{0<t'\leq N}\frac{\delta(t')}{P(t'-1)} \leq \frac{\ln n}{\varepsilon} + (1+3\varepsilon)\frac{||\mathbf{f}^{(N)}||_1}{P(t^*)}$$

The second inequality uses the fact that $||f^{(N)}||_1 = \sum_{t'}\delta(t')$ and that $P(t^*) \leq P(t)$ for all t.

Claim A.8. $\mathsf{cong}(\boldsymbol{f}^{(N)}) \geq N/n$*, so this implies that* $\mathsf{cong}(\boldsymbol{f}^{(N)}) \geq \ln n/\varepsilon^2$ *when* $N \geq n\ln n/\varepsilon^2$.

Proof. We will argue that $\sum_{i\in[n]}\frac{\mathbf{f}^{(t)}A}{c_i}$ increases by at least one on each day. Since this sum is at most $n\mathsf{cong}(\mathbf{f}^{(t)})$, we have the desired result. To see the increase, let i be the column that defines $\delta(t)$, that is $i = \arg\min_{i\in[n]} c_i/A_{j(t),i}$. Notice that $((\mathbf{f}^{(t+1)})^TA)_i = ((\mathbf{f}^{(t)})^TA)_i + \delta(t)A_{j(t),i} \geq ((\mathbf{f}^{(t)})^TA)_i + c_i$. This shows an increase of one in the above sum. □

Plugging in this term, we have that:

$$\mathsf{cong}(\mathbf{f}^{(N)}) \leq \varepsilon\mathsf{cong}(\mathbf{f}^{(N)}) + (1+3\varepsilon)\frac{||\mathbf{f}^{(N)}||_1}{P(t^*)}$$

This implies that $P(t^*) \leq (1+6\varepsilon)D(N)$.

A.4. Fast LP solver (Proof of Theorem 2.23)

By Lemma 2.17, it is enough to solve kECSS LP with KC inequalities.

A.4.1. Interpretation of MWU Framework

We interpret the analysis in Appendix A.3 in the language of graphs. An interesting feature is that the dual variables are only used in the analysis; it is not used in the implementation at all.

We use $\mathbf{w}^{\text{mwu}}$ to be the weights that the primal LP maintains. Let $\{(C^{(t)}, F^{(t)}, c^{(t)}_{\text{min}})\}_{t\leq T}$ a sequence of normalized free cuts $(C^{(t)}, F^{(t)})$ and the value $c^{(t)}_{\text{min}} = \min_{e\in C^{(t)}\setminus F^{(t)}} c(e)$ obtained by the MWU algorithm up to day T. For each edge e, we define congestion $\mathsf{cong}(e) = \frac{1}{c(e)} \cdot \sum_{t\leq T\colon e\in C^{(t)}\setminus F^{(t)}} c^{(t)}_{\text{min}}$. The congestion of the graph is denoted as $\mathsf{cong}(G) = \max_{e\in E} \mathsf{cong}(e)$. Note that $\mathsf{cong}(G)$ is precisely the same as cong in Equation (A.1) when we restrict the LP instance to kECSS LP. Furthermore, by definition, we have

$$\forall e, \mathbf{w}^{\text{mwu}}(e) \leq \frac{1}{c(e)} \cdot \exp(\varepsilon\mathsf{cong}(G)) \tag{A.2}$$

Since the running time of the Range Punisher depends on the change of weights, we need to ensure that the total change (the sum-of-log (SOL) terms) is at most near-linear. We bound the SOL term using a slightly different stopping criteria: Observe that the analysis rely crucially on the fact that congestion $\mathsf{cong}(G) \geq \frac{1}{\varepsilon^2}\ln m$. We could also use $\mathsf{cong}(G) \geq \frac{1}{\varepsilon^2}\ln m$ as a stopping condition (instead of running up to $O(\frac{1}{\varepsilon^2}m\log m)$ days), and the stopping condition implies the number of days is at most $O(\frac{1}{\varepsilon^2}m\log m)$.

We can infer $\mathsf{cong}(G)$ from the weight function $\mathbf{w}^{\text{mwu}}$ by the following. Let $\phi^{\text{mwu}}(e) := \frac{1}{\varepsilon} \cdot \ln(c(e) \cdot \mathbf{w}^{\text{mwu}}(e))$ for all $e \in E$. By definition of $\mathsf{cong}(e)$, we have $\mathbf{w}^{\text{mwu}}(e) = \frac{1}{c(e)} \cdot \exp(\varepsilon\mathsf{cong}(e))$, and so $\phi^{\text{mwu}}(e) = \mathsf{cong}(e)$. Therefore, we have

$$\|\phi^{\text{mwu}}\|_\infty = \mathsf{cong}(G). \tag{A.3}$$

Algorithm 9: kECSSLPSolver(G, c, ε)

```
    Input: An undirected graph $G = (V, E)$, a cost function $c$, $\varepsilon \in (0, 1)$
    Output: A fractional solution $\mathbf{w}^{\text{sol}}$.
 1  $\forall e \in E, \mathbf{w}^{\text{mwu}}(e) \leftarrow \frac{1}{c(e)}$
 2  Let $\tilde{\lambda}$ be an $(1+\varepsilon)$-approximation to $\mathsf{OPT}_{\mathbf{w}^{\text{mwu}}}$
 3  $\lambda \leftarrow \frac{\tilde{\lambda}}{1+\varepsilon}$
 4  $\mathbf{w}^{\text{best}} \leftarrow \frac{\mathbf{w}^{\text{mwu}}}{\tilde{\lambda}}$
 5  repeat
 6      $(\mathbf{w}^{\text{mwu}}, \mathbf{w}^{\text{sol}}) \leftarrow$ RangePunish$(G, \mathbf{w}^{\text{mwu}}, \lambda)$
 7      $\lambda \leftarrow \lambda(1+\varepsilon)$
 8      if $c^T \boldsymbol{w}^{\text{best}} > c^T \boldsymbol{w}^{\text{sol}}$ then $\mathbf{w}^{\text{best}} \leftarrow \mathbf{w}^{\text{sol}}$.
 9  until $\exists$ a day such that $\|\phi\|_\infty > \frac{1}{\varepsilon^2} \cdot \ln m$ (and early terminate)
10  return $\boldsymbol{w}^{\text{best}}$.
```

A.4.2. Algorithm

For the implementation, recall that we denote $\mathbf{w}^{\text{mwu}}$ to be the real weights on MWU framework, and $\mathbf{w}$ to be the approximate weight that the data structure maintains.

We describe extra bookkeeping from RangePunisher to construct to the final solution. First, it outputs a pair of weight function $(\mathbf{w}^{\text{mwu}}, \mathbf{w}^{\text{sol}})$ where $\mathbf{w}^{\text{mwu}}$ is the weights at the end of RangePunisher and $\mathbf{w}^{\text{sol}} = \frac{\mathbf{w}^{\text{init}}}{\mathsf{val}_{\mathbf{w}^{\text{init}}}(C,F)}$ where $\mathbf{w}^{\text{init}}$ is the initial weight function for RangePunisher, and (C, F) is the first normalized mincut obtained during the range punisher.

Since the range punisher maintains approximate weights, we next explain how to detect the stopping condition using approximate weights. We want to stop as soon as $\|\phi^{\text{mwu}}\|_\infty > \frac{1}{\varepsilon^2} \cdot \ln m$. Since we can only keep the approximate weights, we can only detect the approximate value with $O(1/\varepsilon)$-additive error as follows. First, it keeps track of $\phi(e) := \frac{1}{\varepsilon} \cdot \ln(c(e) \cdot \mathbf{w}(e))$ for all $e \in E$, and early stop as soon as $\|\phi\|_\infty > \frac{1}{\varepsilon} \cdot \ln m$. Since $\mathbf{w}$ is $(1+\varepsilon)$-approximation to the real weight $\mathbf{w}^{\text{mwu}}$, it implies that with respect to weight right before the stopping day, $\|\phi^{\text{mwu}}\|_\infty \leq \frac{1}{\varepsilon} \cdot \ln m + O(\varepsilon^{-1}) = O(\frac{1}{\varepsilon} \ln m)$.

The algorithm for LP solver is described in Algorithm 9.

A.4.3. Correctness

We first show that Algorithm 9 punish a sequence of $(1+O(\varepsilon))$-approximate normalized free cuts with respect to $\mathbf{w}^{\text{mwu}}$ where the weight update rule is defined in the PunishMin operations. Initially, $\mathbf{w}^{\text{mwu}}(e) = \frac{1}{c(e)}$ for all $e \in E$. By definition, $\mathsf{OPT}_{\mathbf{w}^{\text{mwu}}} \in [\tilde{\lambda}/(1+\varepsilon), \tilde{\lambda})$ and thus $\mathsf{OPT}_{\mathbf{w}^{\text{mwu}}} \in [\lambda, (1+\varepsilon)\lambda)$. For each iteration where $\mathsf{OPT}_{\mathbf{w}^{\text{mwu}}} \in [\lambda, (1+\varepsilon)\lambda)$, the range punisher (Theorem 2.22) keeps punishing $(1+O(\varepsilon))$-approximate normalized free cuts until $\mathsf{OPT}_{\mathbf{w}^{\text{mwu}}} \geq (1+\varepsilon)\lambda$.

By discussion in Appendix A.4.1, and Theorem A.6, there must be a day t^* such that in some range such that $\frac{w^{(t^*)}}{\mathsf{val}_{w^{(t^*)}}(C^{(t^*)}, F^{(t^*)})}$ is $(1+O(\varepsilon))$-approximation to the LP solution where $w^{(t^*)}$ is $\mathbf{w}^{\text{mwu}}$ at day t^*. Since each normalized cut value is within $(1+\varepsilon)$ factor from any other cut inside the same range, we can easily show that the first cut in the range is $(1+\varepsilon)$-competitive with *any* cut in the range. Therefore, Algorithm 9 outputs $(1+O(\varepsilon))$-approximate solution to kECSS LP.

A.4.4. Running Time

By Corollary 2.8, the running time for computing the value $\tilde{\lambda}$ is $\tilde{O}(\frac{1}{\varepsilon} \cdot m)$. By Theorem 2.22, the total running time is

$$\widetilde{O}(m\ell + K + \frac{1}{\varepsilon} \cdot \sum_{e \in E} \log(\frac{\mathbf{w}^{\text{mwu}}(e)}{\mathbf{w}^{\text{init}}(e)})),$$

where ℓ is the number of iterations, and K is the total number of normalized free cuts punished (including all iterations), $\mathbf{w}^{\text{mwu}}$ is the final weight at the end of the algorithm, and $\mathbf{w}^{\text{init}}(e) = 1/c(e)$ for all e.

Since we early stop as soon as $\|\phi\|_\infty > \frac{1}{\varepsilon^2} \cdot \ln m$, it means that the day right before we stop we have $\|\phi^{\text{mwu}}\|_\infty = O(\frac{1}{\varepsilon^2} \cdot \ln m)$. By the stopping condition,

$$\mathsf{cong}(G) \overset{(A.3)}{=} \|\phi^{\text{mwu}}\|_\infty = O(\frac{1}{\varepsilon^2} \cdot \ln m). \tag{A.4}$$

The following three claims finish the proof.

Claim A.9. *$\ell = O(\frac{1}{\varepsilon^2} \log m)$.*

Proof. Initially, we have $\mathsf{OPT}_{\mathbf{w}^{\text{mwu}}} \in [\lambda, (1+\varepsilon)\lambda)$. By Equation (A.2), we have $\mathbf{w}^{\text{mwu}}(e) \leq \frac{1}{c(e)} \cdot \exp(\varepsilon \mathsf{cong}(G)) \overset{(A.4)}{=} O(\frac{1}{c(e)} \cdot m^{O(\frac{1}{\varepsilon})})$ for all $e \in E$. Let $(C^{(0)}, F^{(0)})$ be the first normalized free cut that we punish. Let λ_0 be the value of that cut. We have that each edge is increase by at most a factor of $m^{O(\frac{1}{\varepsilon})}$, and thus the cut at day right before the stopping happens must be smaller than $\lambda_0 \cdot m^{O(\frac{1}{\varepsilon})}$. Therefore, the number of ranges is $\log_{1+\varepsilon}(m^{O(\frac{1}{\varepsilon})}) = O(\frac{1}{\varepsilon^2} \log m)$. □

Claim A.10. *$K = O(\frac{1}{\varepsilon^2} m \log m)$.*

Proof. Observe that for each normalized free cut (C, F) that we punish there exists a bottleneck edge $e \in C \setminus F$ whose $c(e)$ is minimum. By the weight update rule, the congestion is this edge is increased by exactly 1. Therefore, the number of normalized free cuts is at most $O(m \cdot \mathsf{cong}(G)) \overset{(A.4)}{=} O(\frac{1}{\varepsilon^2} m \log m)$. □

Claim A.11. *For each e, $\log(\frac{\boldsymbol{w}^{\text{mwu}}(e)}{\boldsymbol{w}^{\text{init}}(e)})) = O(\frac{1}{\varepsilon} \log m)$.*

Proof. Recall that the initial weight $\mathbf{w}^{\text{init}}(e) = 1/c(e)$ for all e. Therefore,

$$\forall e \in E, \log(\frac{\mathbf{w}^{\text{mwu}}(e)}{\mathbf{w}^{\text{init}}(e)})) \overset{(A.2)}{\leq} \varepsilon \mathsf{cong}(G) \overset{(A.4)}{\leq} O(\frac{1}{\varepsilon} \cdot \log m).$$

□

APPENDIX B

Dynamic Spanner

B.1. Lifting the Machine-Disjoint Assumption

In the proof of Lemma 3.14 in Section 3.4, we assume Assumption 3.17 which says that, for a fixed job u, routines for u are machine-disjoint. In this section, we show how to remove it.

Reiterating the pain point. Each experiment X_i is defined **with respect to** a sampling on each edge. In any sequence $\mathcal{X}^{(t,x)}$, there could be two (t,x)-relevant experiments $X_i^{(t,x)}$ and $X_j^{(t,x)}$ where $i < j$ such that $t(X_i^{(t,x)}) = t(X_j^{(t,x)})$ and $\text{job}(X_i^{(t,x)}) = \text{job}(X_j^{(t,x)})$. By our sampling process, these two variables are negatively correlated as knowing $X_i^{(t,x)} = 1$ would immediately imply that $X_j^{(t,x)} = 0$ and vice versa.

To see why the condition in Lemma 3.22 may not be true in this case, let consider the case where we have two random variables X and Y where $\mathbb{P}[X = 1] = p$ be the probability that X being 1. If these two variables are correlated in such a way that $X + Y = 1$, then we cannot show that $\hat{X}, \hat{Y}$ such that $X + Y \preceq \hat{X} + \hat{Y}$ using Lemma 3.22 unless $\mathbb{P}[\hat{X}] = \mathbb{P}[\hat{Y}] = 1$. We want to make sure that variables in the sequence we want to analyze do not have such correlation.

Proving strategy. Here, we will define two other sequences $\mathcal{Y}^{(t,x)}$ and $\hat{\mathcal{Y}}^{(t,x)}$. These two sequences are defined analogous to $\mathcal{X}^{(t,x)}$ and $\hat{\mathcal{X}}^{(t,x)}$. We then define $\widetilde{\deg}_{A^t}(x) = \sum_{\hat{Y} \in \hat{\mathcal{Y}}^{(t,x)}} \hat{Y}$. We will replace $\widehat{\deg}_{A^t}(x)$ in Key Steps 2 and 3 by $\widetilde{\deg}_{A^t}(x)$. More precisely, instead of Lemma 3.20, we will prove $\overline{\deg}_{A^t}(x) \preceq \widetilde{\deg}_{A^t}(x)$, and we will prove that, w.h.p., $\widetilde{\deg}_{A^t}(x) \leq 2\text{target}^t(x) + O(\log|M|)$ instead of Lemma 3.21.

Redefining relevant experiments. For a fixed $\mathcal{X}^{(t,x)}$, we let $\pi^{(t,x)}(t',u) = \{X_i^{(t,x)} : t(X_i^{(t,x)}) = t', \text{job}(X_i^{(t,x)}) = u\}$ be the set of (t,x)-relevant experiments containing u taken at time t'. We then let $Y^{(t,x)}(t',u)$ be a boolean random variable that is one if any of experiments in $\pi^{(t,x)}(t',u)$ succeed. In other words,

$$Y^{(t,x)}(t',u) = \sum_{X \in \pi^{(t,x)}(t',u)} X.$$

Note that $Y^{(t,x)}(t',u)$ is a boolean random variable because only at most one variable in $\pi^{(t,x)}(t',u)$ can be true as they are being sampled with the same $\textsc{Resample}(u)$ call. We say that $Y^{(t,x)}(t',u)$ is **well-defined** if $|\pi^{(t,x)}(t',u)| > 0$. Note that

$$\mathbb{P}[Y^{(t,x)}(t',u) = 1] = \frac{|\pi^{(t,x)}(t',u)|}{\deg_{G^{t'}}(u)}.$$

Now let us define more notations to help count the number of relevant experiments in form of $Y^{(t,x)}(.,.)$.

Definition B.1. *We say that a timestep t' is (t,x,u)-relevant if $Y^{(t,x)}(t',u)$ is well-defined. Let $Rel(t,x,u)$ be the number of (t,x,u)-relevant timesteps, this is exactly the number of possible t' such that $Y^{(t,x)}(t',u)$ is well-defined. We also let $\widetilde{Rel}(t,x) = \sum_u Rel(t,x,u)$ be the number of possible pairs of t',u such that $Y^{(t,x)}(t',u)$ is well-defined.*

We then define a sequence $\mathcal{Y}^{(t,x)} = Y_1^{(t,x)}, Y_2^{(t,x)}, \ldots, Y_{\widetilde{Rel}(t,x)}^{(t,x)}$ to be the sequence of well-defined $Y^{(t,x)}(.,.)$ order by the time of these experiments. Notice that $\overline{\deg}_{A^t}(x) = \sum_{X \in \mathcal{X}^{(t,x)}} X = \sum_{Y \in \mathcal{Y}^{(t,x)}} Y$.

Now we discuss the replacement of $\hat{\mathcal{X}}^{(t,x)}$. For each $Y_i^{(t,x)}$, we let $\hat{Y}_i^{(t,x)}$ be an independent random variable that is true with probability $\frac{|\pi^{(t,x)}(t',u)|}{\deg_{G^t}(u)}$. Then the sequence $\hat{\mathcal{Y}}^{(t,x)}$ is $\hat{Y}_1^{(t,x)}, \hat{Y}_2^{(t,x)}, \ldots, \hat{Y}_{\widetilde{Rel}(t,x)}^{(t,x)}$. As mentioned above, $\widetilde{\deg}_{A^t}(x) = \sum_{\hat{Y} \in \hat{\mathcal{Y}}^{(t,x)}} \hat{Y}$.

We state the two main claims below.

Claim B.2 (Key Step 2′)**.** $\overline{\deg}_{A^t}(x) \preceq \widetilde{\deg}_{A^t}(x)$.

Proof. As $\overline{\deg}_{A^t}(x) = \sum_{Y \in Y^{(t,x)}} Y$, by Lemma 3.22, we need to show that $\mathbb{P}[Y_i^{(t,x)} = 1 | Y_1^{(t,x)}, \ldots, Y_{i-1}^{(t,x)}] \leq \mathbb{P}[\hat{Y}_i^{(t,x)}]$. By replacing $\mathcal{X}^{(t,x)}$ and $\hat{\mathcal{X}}^{(t,x)}$ with $\mathcal{Y}^{(t,x)}$ and $\hat{\mathcal{Y}}^{(t,x)}$, the proof here can be done with the same arguments we used in Section 3.5.1. Note that the proof works because random variables from $\mathcal{Y}^{(t,x)}$ are *not* negatively correlated. □

Claim B.3 (Key Step 3′)**.** $\widetilde{\deg}_{A^t}(x) \leq 2\log(t)\text{target}^t(x) + O(\log|M|)$ *with probability* $1 - 1/|M|^{10}$.

Proof. We can follow exactly the proof in Section 3.5.2. The only crucial part is to show that $\mathbb{E}[\widetilde{\deg}_{A^t}(x)] = \mathbb{E}[\widehat{\deg}_{A^t}(x)]$. The other parts can be argued exactly the same way.

This is true by the way we define the process. For each $\hat{Y}_i^{(t,x)}$ that is being 1 with probability $c/\deg_{G^t}(\text{job}(Y_i^{(t,x)})$ for some integer c, we have c different boolean variables $\hat{X}_1, \hat{X}_2, \ldots, \hat{X}_c$ in $\hat{\mathcal{X}}^{(t,x)}$ that is being 1 with probability $1/\deg_{G^t}(\text{job}(Y_i^{(t,x)}))$. Hence, the two expectations are identical by linearity of expectation. □

B.2. A Fully-dynamic-to-decremental Reduction for Spanners

In this section, we give a reduction from a fully-dynamic spanner to a decremental spanner. This reduction is due to [BKS12] and we provide it here for the completeness of our paper.

Let $E_1 \ldots E_j$ be a partition of E, then the observation below states that the union of spanners of $E_1 \ldots E_j$ is a spanner of E.

Observation B.4 (Observation 5.2 in [BKS12])**.** *For a given graph $G = (V, E)$, let $E_1 \ldots E_j$ be a partition of the set of edges E, and let $\mathcal{E}_1 \ldots \mathcal{E}_j$ be respectively the t-spanner of subgraphs $G_1 = (V, E_1), \ldots, G_j = (V, E_j)$. Then $\bigcup_i \mathcal{E}_i$ is a t-spanner of the original graph $G = (V, E)$.*

With this observation, the idea behind the reduction is to split into $O(\log n)$ subgraphs in such a way that every subgraph, excepts one subgraph, is a decremental instance.

Formally, let ℓ_0 be the greatest integer such that $2^{\ell_0} \leq n^{1+1/k}$. We do the following.

(1) We partition E into $E_0 \dots E_j, j = \lceil \log_2 n^{1-1/k} \rceil$ such that $|E_i| \leq 2^{\ell_0+i}$. Each edge will belong to only one set E_i and we keep track of this information.

(2) For each E_i, $i > 0$, we maintain $H_i = (V, \mathcal{E}_i)$, which is a $(2k-1)$-spanner of (V, E_i).

(3) We maintains a binary counter $\mathbf{C}$ which counts from 0 to $\frac{n(n-1)}{2}$. This will be used to decide when to rebuild E_i.

In the beginning, we set $E_j = E$ and $E_i = \emptyset$ for all $i < j$. The counter $\mathbf{C}$ is set to 0. Any edge deletion of $e \in E_i$ for any i is handled as in the decremental case. When an edge e is inserted, we increment the counter $\mathbf{C}$ by one. Let g be the highest bit of $\mathbf{C}$ that gets flipped. If $g \leq \ell_0$, then we put e in E_0 and $\mathcal{E}_0$. Otherwise, we insert e into E_h, where $h = g - \ell_0$, move all edges from E_i, $i < h$ to E_h. At this moment $E_i = \emptyset$ for all $i < h$. We then rebuilt the spanner $\mathcal{E}_h$.

From Observation B.4, $\bigcup_i \mathcal{E}_i$ is a $(2k-1)$-spanner of G.

Lemma B.5 (Restate Lemma 3.6)**.** *Suppose that for a graph G with n vertices and m initial edges undergoing only edge deletions, there is an algorithm that maintains a $(2k-1)$-spanner H of size $O(S(n))$ with $O(F(m))$ total recourse where $F(m) = \Omega(m)$, then there exists an algorithm that maintains a $(2k-1)$-spanner H' of size $O(S(n) \log n)$ in a fully dynamic graph with $O(F((U) \log n))$ total recourse. Here U is the number of updates made throughout the algorithm, starting from an empty graph.*

Proof of Lemmma 3.6. We use the reduction above to partition the graph into $E_1, \dots, E_j$. We then use the decremental algorithm to maintain each $\mathcal{E}_i$ for all $i > 0$.

We first show that the size of $|H'| = |\bigcup_i \mathcal{E}_i| = O(\log n S(n))$. As we have $O(\log n)$ subgraphs, $|\bigcup_i \mathcal{E}_i| = \sum_i S(|E_i|) \leq O(S(n) \log n)$.

Now we show the total recourse. Let $\mathcal{G}_1, \mathcal{G}_2, \dots, \mathcal{G}_k$ be all the graph we rebuilt throughout all the timesteps. Then the total recourse is bounded by $\sum_i F(|\mathcal{G}_i|) \leq F(\sum_i |\mathcal{G}_i|)$. Notice that the level of any edge e can only go up, so e can contribute the recourse to only $\log n$ different graphs. Hence, this inequality becomes

$$F(\sum_i |\mathcal{G}_i|) \leq F((U)\log n).$$

$\square$

B.3. Missing Proofs from Section 3.2

Claim B.6 (Restate Claim 3.7)**.** *The subgraph $H = (V, E_1 \cup E_2 \cup E_3)$ is a 3-spanner of G consisting of at most $O(n\sqrt{n})$ edges.*

Proof. We need to show that (1) H is a 3-spanner and (2) $E_H = E_1 \cup E_2 \cup E_3$ has size at most $O(n\sqrt{n})$. GG

Stretch. Consider an edge $e = (u, v)$ where $u \in V_i, v \in V_j$. We show that H has a path of length at most 3 between u and v. The easy case is when $(u, v) \in E_H$. This gives us a path of one edge. It happens when $u = c_i(v)$ or $v = c_j(u)$, or $i = j$. Suppose $(u, v) \notin E_H$. Consider $v' = c_j(u)$. Since u is a common neighbor between v and v', $P(v, v')$ is not empty. As $e \notin E_H$, $u \neq w_{vv'}$. As, the path $u, v', w_{vv'}, v$ has length exactly 3, the stretch part is concluded.

Size. Each vertex u has upto $\sqrt{n}$ partners. Since we have n vertices, $|E_1| = O(n\sqrt{n})$. For E_2, the graph induced on V_i has at most $O(n)$ edges. Since we have $\sqrt{n}$ buckets, $|E_2| = O(n\sqrt{n})$. For E_3, $|E_3|$ is bounded by the number of witnesses we need. Since we have $O\sqrt{n}$ buckets, and we have $O(n)$ pairs of vertices within the same bucket, for all buckets, $|E_3|$ must be bounded by $O(n\sqrt{n})$. We conclude the proof by saying that $|E_H| = O(|E_1| + |E_2| + |E_3|) = O(n\sqrt{n})$

$\square$

B.3.1. Making the update time worst case

It is evident that the update time in Lemma 3.10 is in worst-case. The whole algorithm is amortized only because we are *replacing* E_3 from scratch at the start of each phase, which takes $\tilde{O}(n^2)$ time, and this time needs to be amortized over the length of the phase. Using very standard techniques from the existing literature on dynamic algorithms (see e.g. [Tho05, BCCK16, NSW17, Kis21]), we can easily convert this into an overall worst case update time guarantee. The idea is to *spread out* the $\tilde{O}(n^2)$ cost of rebuilding at the start of a given phase through a sufficiently large chunk of updates in the phase preceding it.

For completeness, we will describe the idea in more detail here. Let G_i be the graph after i updates. We will use one instance of our algorithm to handle $L = \tilde{\Theta}(n\sqrt{n})$, that is, the i-th instance will be used to handle the time steps $[(i-1)L, iL)$. For our idea to work, any copy must be able to handle $2L$ updates (which is fine in our case). At the beginning, we initiate the first copy D_1 with time $\tilde{O}(n^2)$. We want to initiate D_2, as well as feed D_2 with L updates, so that D_2 is ready to use at the timestep L. This can be done in the following manner.

- During time steps $[0, L/3)$, we initiate D_2 with the graph G_0,
- During time steps $[L/3, 2L/3)$, we carefully feed the surviving[1] output from D_2 into our actual output,
- During time steps $[2L/3, L)$, we update D_2 with updates from time steps $[0, L)$, 3 updates at a time.

Hence, at time step L, we can switch from D_1 to D_2, disregard D_1, and start initiating D_3. More generally, suppose at time step iL, after we have initiated D_i, which we will use for time step $[iL, (i+1)L)$. In the following L time steps, to disregard D_{i-1} and initiate D_{i+1}, we do the following.

[1]Some edges in the spanner of D_2 might be deleted between the time step $[0, 2L/3]$, we must not add deleted edges in our actual output.

- During time steps $[iL, iL + L/3)$, we initiate D_{i+1} with the graph G_{iL} and slowly disregard the output from D_{i-1},
- During time steps $[iL+L/3, iL+2L/3)$, we carefully feed the surviving output from D_{i+1} into our actual output,
- During time steps $[iL + 2L/3, (i + 1)L)$, we update D_{i+1} with updates from time steps $[iL, (i + 1)L)$, 3 updates at a time.

Then, at time step $(i+1)L$, we completely disregard D_{i-1}, and completely initiate D_{i+1}, hence maintaining all the desired properties. Our actual output at time step t, is consisting of E_{D_i} and $E_{D_{i+1}}$, which are output of D_i and D_{i+1}, respectively. Notice that, (V, E_{D_i}) is a 3-spanner of G_t, hence, $(V, E_{D_i} \cup E_{D_{i+1}})$ is also a 3-spanner of G_t. This is because a spanner with extra edges is still a spanner. Hence, as long as $|E_{D_i} \cup E_{D_{i+1}}|$ is not too large, we can keep both of them in our output at the same time. We conclude by saying that, this idea of maintaining multiple copies, along with our Lemma 3.11, implies Theorem 3.3.

Note. As a corollary, we note that the idea above does not have anything to do with Spanner. Rather, it is applicable to any dynamic problem, as long as, it can handle batch updates in worst-case time, and that the bottleneck of the algorithm is on the initialization time.

APPENDIX C

Graph Versioning

C.1. DP on tree via FPTAS 4.4.1

C.1.1. Reduction from general tree to binary tree

Lemma C.1. *If algorithm $\mathcal{A}$ solves* BMR *on binary tree instances in $O(f(n))$ time where n is the number of vertices in the tree, then there exists algorithm $\mathcal{A}'$ solving* BMR *on all tree instances in $O(f(2n))$ time.*

Proof Sketch. If a node v has more than two children, we modify the graph as follows:

(1) Create node v' and attach it as a child of v.
(2) Move all but the left-most children of v to be children of v'
(3) Set the deltas of $(v, v') = (v', v) = 0$; set $(v', c_i) = (v, c_i)$ and $(c_i, v') = (c_i, v)$ for all transferred children c_i.

By repeating this process we obtain a binary tree with $\leq 2n$ nodes which has the same optimal objective value as before. Hence, after producing a binary tree, we can utilize the algorithm for binary tree to solve BMR on any tree. □

C.1.2. All connection cases for DP for MSR on trees

We present the 5 cases in the recurrent step here as promised in Section 4.4.1. All other cases are symmetric to the cases we present, hence omitted. We use S_i to denote the minimum storage cost in case i, as shown in Fig. 4.7.

$$S_1 = s_v + \min_{\rho_1+\rho_2=\rho} \Big\{ \min_{k_1,\gamma_1}\{DP[c_1][k_1][\gamma_2][\rho_1]\} + \min_{k_2,\gamma_2}\{DP[c_2][k_2][\gamma_2][\rho_2]\Big\}$$

$$S_3 = s_v + s_{v,c_2} - s_{c_2} + \min_{\rho_1+\rho_2=\rho} \Big\{ \min_{k',\gamma_1}\{DP[c_1][k'][\gamma_1][\rho_1]\} + DP[c_2][k-1][0][\rho_2 - (k-1)r_{v,c_2}]\Big\}$$

$$S_4 = s_v + s_{v,c_1} - s_{c_1} + s_{v,c_2} - s_{c_2} + \min_{\rho_1+\rho_2=\rho} \min_{k_1+k_2=k-1} \Big\{DP[c_1][k_1][0][\rho_1 - k_1 r_{v,c_1}] + DP[c_2][k_2][0][\rho_2 - k_2 r_{v,c_2}]\Big\}$$

$$S_6 = s_{c_2,v} + \min_{\rho_1+\rho_2=\rho} \Big\{ \min_{k_2}\{DP[c_2][k_2][\gamma - r_{c_2,v}][\rho_2 - \gamma]\} + \min_{k_1,\gamma'}\{DP[c_1][k_1][\gamma'][\rho_1]\}\Big\}$$

$$S_7 = s_{c_2,v} + s_{v,c_1} - s_{c_1} + \min_{\rho_1+\rho_2=\rho} \Big\{DP[c_1][k-1][0][\rho_1 - (k-1)\cdot(r_{v,c_1} + \gamma)] + \min_{k'}\{DP[c_2][k'][\gamma - r_{c_2,v}][\rho_2 - \gamma]\}\Big\}$$

C.2. Supplementary materials for Section 4.4.3

C.2.1. Algorithms in Section 4.4.3

We present the pseudo code for Algorithms 10, 11, and 12 below, as mentioned in Section 4.4.3:

C.2.2. Calculation of ρ

We hereby demonstrate that the method for calculating ρ_Δ in Algorithm 13 is indeed correct.

For a pair of compatible partial solutions $\mathcal{T}_a, \mathcal{T}_b$ with regards to $\mathcal{T}_z$, ρ_Δ is defined such that $\rho_a + \rho_b = \rho_z - \rho_\Delta$. Therefore, as we go down a path described by $\mathcal{T}_z$ in topological

Algorithm 10: Scan Uprooted Nodes

```
Input : $S_z, S_a, S_b, \mathcal{T}_z$
1  $U_a \leftarrow \varnothing$; $U_b \leftarrow \varnothing$
2  for $v$ in $S_z$ do
3      if $v$ in $S_a$ and $\mathrm{Par}_z(v)$ in $S_b \setminus S_a$ then
4          $U_a \leftarrow U_a \cup \{v\}$
5      else if $v$ in $S_b$ and $\mathrm{Par}_z(v)$ in $S_a \setminus S_b$ then
6          $U_b \leftarrow U_b \cup \{v\}$
7  return $U_a, U_b$
```

Algorithm 11: Un-Uproot

```
Input : $S_z, \mathcal{T}_z, S_a, U_a, S_b, U_b$
1  $\mathcal{T}_a := (\mathrm{Par}_a, \mathrm{Dep}_a, \mathrm{Ret}_a, \mathrm{Anc}_a) \leftarrow \mathcal{T}_z$
2  $\mathcal{T}_b := (\mathrm{Par}_b, \mathrm{Dep}_b, \mathrm{Ret}_b, \mathrm{Anc}_b) \leftarrow \mathcal{T}_z$
3  Sort $S_z$ in topological order according to $\mathrm{Anc}_z$
4  for $v \in S_z$ do
5      if $v \in U_a$ then                                   /* Case 1 in Fig. 4.9 */
6          $\mathrm{Par}_a(v) \leftarrow v$; $\mathrm{Ret}_a(v) \leftarrow 0$; $\mathrm{Anc}_a(v) \leftarrow \varnothing$
7          for $u \in \mathrm{Anc}_b(v)$ do
               // Dependents of $v$, including $v$, are removed.
8              $\mathrm{Dep}_b(u) \leftarrow \mathrm{Dep}_b(u) - \mathrm{Dep}_b(v) + 1$
9      else if $v \in U_b$ then                              /* Case 3 in Fig. 4.9.   */
10         $\mathrm{Par}_b(v) \leftarrow v$; $\mathrm{Ret}_b(v) \leftarrow 0$; $\mathrm{Anc}_b(v) \leftarrow \varnothing$
11         for $u \in \mathrm{Anc}_a(v)$ do
12             $\mathrm{Dep}_a(u) \leftarrow \mathrm{Dep}_a(u) - \mathrm{Dep}_a(v) + 1$
13     else                                                  /* Case 2 in Fig. 4.9.  */
           // Do nothing if $v \notin S_a$.  Same for the following lines.
14         $\mathrm{Anc}_a(v) \leftarrow \mathrm{Anc}_a(\mathrm{Par}_z(v)) \cup \{\mathrm{Par}_z(v)\}$
15         $\mathrm{Anc}_b(v) \leftarrow \mathrm{Anc}_b(\mathrm{Par}_z(v)) \cup \{\mathrm{Par}_z(v)\}$
16         $\mathrm{Ret}_a(v) \leftarrow \mathrm{Ret}_a(\mathrm{Par}_z(v)) + r_{\mathrm{Par}_z(v),v}$
17         $\mathrm{Ret}_b(v) \leftarrow \mathrm{Ret}_b(\mathrm{Par}_z(v)) + r_{\mathrm{Par}_z(v),v}$
18 return $\mathcal{T}_a, \mathcal{T}_b$
```

order, we analyze how many times the retrieval cost of an edge is counted by both ρ_a and ρ_b as compared to that by ρ_z. For example, in figure C.1, the retrieval cost of edge $(1, 2)$ is counted 8 times in $\mathcal{T}_z$, zero times in $\mathcal{T}_a$, and twice in $\mathcal{T}_b$. The details are as below:

(1) We observe that all edges in $\mathcal{T}_a$ and $\mathcal{T}_b$ are must also be in $\mathcal{T}_z$: in Compatibility, no additional edges. Hence, it suffices to focus on all edges of $\mathcal{T}_z$

Algorithm 12: COMPATIBILITY

```
   Input  : $S_z, S_a, S_b, \mathcal{T}_a, \mathcal{T}_b$
1  $U_a, U_b \leftarrow$ SCAN UPROOTED NOTE$(S_z, S_a, S_b, \mathcal{T}_z)$
2  $\mathcal{T}'_a, \mathcal{T}'_b \leftarrow$UN-UPROOT$(S_z, \mathcal{T}_z, S_a, U_a, S_b, U_b)$
3  for $v \in S_a \cap S_b$ do
4      $\mathrm{ExtDep}_z \leftarrow \mathrm{Dep}_z(v) - \sum_{w \in S_z : \mathrm{Par}_z(w) = v} \mathrm{Dep}_z(w)$      // External dependency.
5      $\mathrm{ExtDep}_a \leftarrow \mathrm{Dep}_a(v) - \sum_{w \in S_a : \mathrm{Par}_a(w) = v} \mathrm{Dep}_a(w)$
6      $\mathrm{ExtDep}_b \leftarrow \mathrm{Dep}_b(v) - \sum_{w \in S_b : \mathrm{Par}_b(w) = v} \mathrm{Dep}_b(w)$
7      if $\mathrm{ExtDep}_z \neq \mathrm{ExtDep}_a + \mathrm{ExtDep}_b$ then
8          return False
9      for $u \in \mathrm{Anc}'_a(v)$ do
10         $\mathrm{Dep}'_a(u) \leftarrow \mathrm{Dep}'_a(u) - (\mathrm{ExtDep}_z - \mathrm{ExtDep}_a)$
           // subtract external dependencies in $V_{[b]} \setminus S_b$ from $\mathcal{T}'_a$
11     for $u \in \mathrm{Anc}'_b(v)$ do
12         $\mathrm{Dep}'_b(u) \leftarrow \mathrm{Dep}'_b(u) - (\mathrm{ExtDep}_z - \mathrm{ExtDep}_b)$
13 if $\mathcal{T}_a = \mathcal{T}'_a$ and $\mathcal{T}_b = \mathcal{T}'_b$ then
14     return True
15 else
16     return False
```

Algorithm 13: DISTRIBUTE RETRIEVAL

```
   Input  : $S_z, \mathcal{T}_z, \rho_z, S_a, S_b, \mathcal{T}_a, \mathcal{T}_b$
1  $\rho_\Delta \leftarrow 0$                                  // We Want $\rho_z - \rho_\Delta = \rho_a + \rho_b$
2  for $v \in S_z$ such that $\mathrm{Par}_z(v) \neq v$ do
3      $\mathrm{Count} \leftarrow \mathrm{Dep}_z(v)$              // the number of times $r_{\mathrm{Par}_z(v),v}$ is counted
4      if $\mathrm{Par}_a(v) = \mathrm{Par}_z(v)$ then
5          $\mathrm{Count} \leftarrow \mathrm{Count} - \mathrm{Dep}_a(v)$
6      if $\mathrm{Par}_b(v) = \mathrm{Par}_z(v)$ then
7          $\mathrm{Count} \leftarrow \mathrm{Count} - \mathrm{Dep}_b(v)$
8      if $\mathrm{Par}_z(v) \in S_z$ then
           // The edge $r_{\mathrm{Par}_z(v),v}$ is over/undercounted.
9          $\rho_\Delta \leftarrow \rho_\Delta + \mathrm{Count} \cdot r_{\mathrm{Par}_z(v),v}$
10     else
           // The entire $\mathrm{Ret}_z(v)$ is over/undercounted.
11         $\rho_\Delta \leftarrow \rho_\Delta + \mathrm{Count} \cdot \mathrm{Ret}_z(v)$
12 return $\rho_\Delta$
```

(2) For each v not materialized in $\mathcal{T}$, we use the temporary variable Count to denote how many times the edge $e = (\mathrm{Par}_z(v), v)$ is over/undercounted in ρ_z.

To put this formally, we can abuse notation and let $\mathrm{Dep}_z(e)$ be the number of times r_e is counted towards total retrieval cost in $\mathcal{T}_z$. Then we have

$$\mathrm{Count} = \mathrm{Dep}_z(e) - (\mathrm{Dep}_a(e) + \mathrm{Dep}_b(e))$$

where if $\mathrm{Par}_a(v) \neq \mathrm{Par}_z(v)$, clearly $\mathrm{Dep}_a(e)$ should be 0, since it's not even stored in $\mathcal{T}_a$.

(3) If both endpoints of e are in S_z, then the amount of retrieval cost overcount in ρ_z is exactly $\mathrm{Count} \cdot r_e$. On the other hand, if e is a delta from outside S_z, the overcount should be $\mathrm{Count} \cdot \mathrm{Ret}_z(v)$, since the entire retrieval cost of v is overcounted Count times.

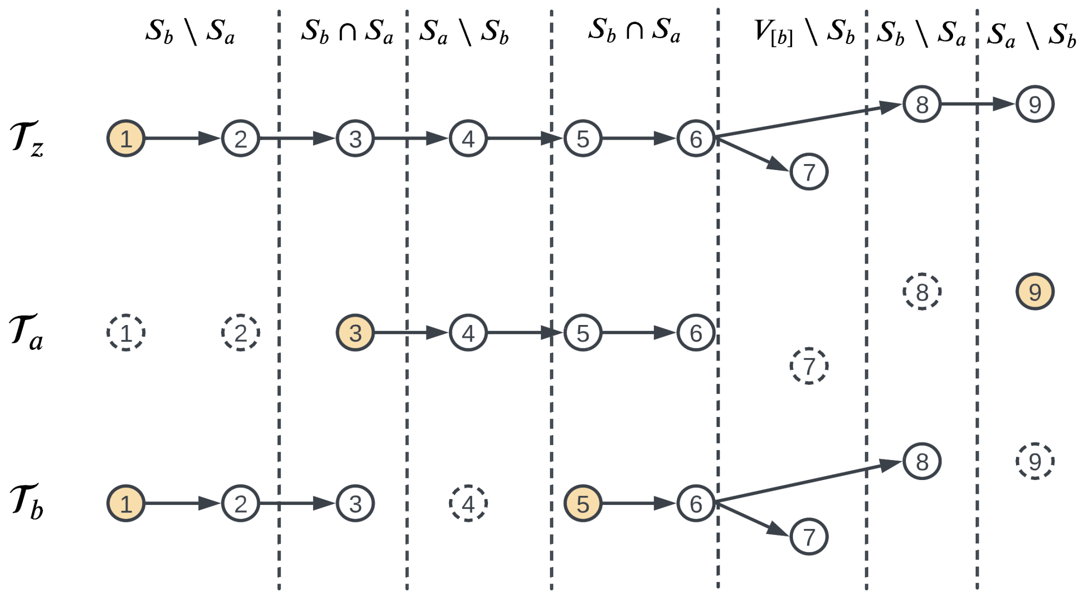


Figure C.1. Illustration of the retrieval path for Fig. 4.8

C.3. ILP Formulation

In the following formulation, we have integer variables $\{x_e\}$ representing how many $v \in V$ is retrieved through the edge e. I_e is a Boolean variable denoting whether edge e is stored. We work on the extend graph with the auxiliary node v_{aux} for convenience.

$$\begin{aligned}
\min \quad & \sum_{e \in E} r_e x_e && && \text{s.t.} \\
& x_e \leq |V-1| I_e && && \text{(indicator constraint)} \\
& \sum_{e \in E} s_e I_e && \leq \mathcal{R} && \text{(storage cost)} \\
& \sum_{e \in In(u)} x_e && = \sum_{e \in Out(u)} x_e + 1 \quad \forall u \in V \setminus \{v_{aux}\} && \text{(sink)} \\
& x_e && \in \{0, 1, \ldots, |V|\} && \\
& I_e && \in \{0, 1\} &&
\end{aligned}$$