ABSTRACT

Title of Dissertation: FAIR AND SCALABLE ALGORITHMS ON MASSIVE GRAPHS

Marina Knittel
Doctor of Philosophy, 2023

Dissertation Directed by: Professor MohammadTaghi Hajiaghayi
Department of Computer Science

The impact of massive data processing on our daily lives is becoming increasingly more clear. With this, we must guarantee that vital, data-driven decision-making systems mitigate the potential negative impacts of the technologies themselves and scale to the massive datasets we have access to today. We explore both of these facets by studying fairness and scalability for algorithms on large graphs.

In Part I, we focus on on fair hierarchical clustering. Our first work on this topic [Ahmadian et al., 2020b] initiates the study of fair hierarchical clustering by extending Chierichetti et al.'s [Chierichetti et al., 2017] notion of representationally proportional flat clustering to the hierarchical setting. From there, we introduce the first approximation algorithms for three well-studied hierarchical clustering optimization problems in the fair context: cost [Dasgupta, 2016], revenue [Moseley and Wang, 2017], and value [Cohen-Addad et al., 2018]. Our initial work studies all three fair optimization problems, and our follow-up works [Knittel et al., 2023a, Knittel et al., 2023b] dive deeper into the notoriously difficult cost optimization problem.

Regarding scalability, we leverage the *Massively Parallel Computation* (MPC) model, as well as its recent successor *Adaptive Massively Parallel Computation* (AMPC), to develop efficient graph algorithms for big data. MPC, discussed in Part II, has been one of the most practically useful models for massively parallel algorithms in the past decade, influencing a number of major frameworks including MapReduce, Hadoop, Spark, and Flume. In this model, we present our work on edge coloring [Behnezhad et al., 2019b], hierarchical clustering [Hajiaghayi and Knittel, 2020], and tree embeddings [Ahanchi et al., 2023].

AMPC improves upon the MPC model by adding access to a distributed hash table while still remaining highly implementable in practice. This allows it to overcome some shortcomings proposed in MPC literature, notably, the 1vs2Cycle Conjecture (i.e., that differentiating between a single cycle and two cycles is difficult in MPC). In Part III, we introduce a highly efficient and general tool for executing tree contractions in AMPC [Hajiaghayi et al., 2022b] and additionally exhibit the power of AMPC on minimum cut problems [Hajiaghayi et al., 2022a].

FAIR AND SCALABLE ALGORITHMS ON MASSIVE GRAPHS

by

Marina Knittel

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2023

Advisory Committee:
Professor MohammadTaghi Hajiaghayi, Chair/Advisor
Professor John Dickerson, Co-Advisor
Dr. Ravi Kumar
Professor David Mount
Professor Alexander Barg

Dedication

For my BC calculus teacher, Radu Toma, and my first best friend, Amy Malzbender.

Rest in peace.

Acknowledgments

Parts of my PhD career were funded by the Ann G Wylie Fellowship and the ARCS Endowment Award.

This work would not have been possible without great collaborative efforts from my coauthors throughout my PhD: Sara Ahmadian, Alexandr Andoni, Soheil Behnezhad, Fotini Christia, Michael Curry, Constantinos Daskalakis, Erik Demaine, Mahsa Derakhshan, Samuel Dooley, Alessandro Epasto, MohammadTaghi Hajiaghayi, Adam Hesterberg, Ravi Kumar, Mohammad Mahdian, Aidan Milliff, Benjamin Moseley, Jan Olkowski, Philip Pham, Hamed Saleh, Max Springer, Hsin-Hao Su, Sergei Vassilvitskii, Yuyan Wang, and Peilin Zhong. I would also like to thank my undergraduate collaborators for contributions to my pre-doctoral work, though not in the scope of this proposal: Jordan R Abrahams, James C Boerkoel Jr, David A Chu, Grace Diehl, Haoxing Du, Jeremy Frank, Gianluca Gross, Ran Libeskind-Hadas, Judy Lin, Nuo Liu, William Lloyd, Ross Mawhorter, Yi Sheng Ong, Reiko Tojo, and Yi-Chieh Wu.

A special thanks goes to my graduate advisors: John P Dickerson and MohammadTaghi Hajiaghayi, my undergraduate research mentors: Yi-Chieh Wu, Eric Allender, James C Boerkoel Jr, Jeremy Frank, and Colleen Lewis, and my future postdoctoral advisors: Barna Saha and Sanjoy Dasgupta.

In addition, I'd like to acknowledge those not mentioned above who have also been a great help throughout my education: Tom Hurst, Bill Gasarch, Chris Kuszmaul, Sharron McElroy,

Mohamed Omar, and Radu Toma.

And finally, the real treasure in a PhD is the friends you make along the way: Aaron, Alex, Alex, Alex, Andrew, Anton, Auguste, Candice, Connor, David, Eddie, Emily, Erica, Ethan, Joy, Kyle, Lillian, Mackenzie, Makana, MG, Noemi, Olive, Nawakhtha, Niall, Pema, Sisi, and Yuelin.

I am the very model of an algorithmic doctorate

I've studied problems randomized, perfect, and approximate

I'm quite good at contracting things, like trees and random spanning trees

Because I know my ABCs, MPCs, AMPCs

My hierarchic clusterings might raise the proletariat

I am the very model of an algorithmic doctorate

Table of Contents

Chapter 1: Introduction

The advent of intelligent processing and inference of massive datasets is what distinguishes modern computing in terms of both potential power and unique challenges. Big data has become a fundamental tool for societal progression, its impacts pervasive across all aspects of our lives. As data continues to grow and this influence becomes more ubiquitous, we must ensure that data systems guarantee a higher standard of efficiency and quality that keeps pace with the times. I believe two of the biggest questions posed by modern computing are as follows: (1) How can we identify, quantify, and mitigate the potential negative impacts of big data in an ever-changing world? (2) How can intelligent technologies keep up with the increasing supply of massive datasets and demand of those affected?

These broad questions go beyond the scope of a single research proposal. My research provides an in depth exploration of multiple topics that will be useful tools in solving both of these problems. In the following sections, I introduce my main areas of research: massively parallel computation, adaptive massively parallel computation, and fair algorithms (specifically, hierarchical clustering). In each section, I summarize the problem, its motivations, its relevance to our overarching research questions, and our related results.

1.1 Fair Hierarchical Clustering

In order to address our first question, we have to understand some of the major shortcomings of current intelligent systems. Some noteworthy examples we are already aware of include: discrimination in allocating health care to racial minorities [Ledford, 2019], the display of ads suggestive of an arrest record more frequently when searching for black-identifying names [Sweeney, 2013], the reflection of systemic biases against minority groups in hiring prediction [Bogen and Rieke, 2018], and the disproportionate assessment that certain races are a higher risk for recidivism [Angwin et al., 2016]. These are some major problems that have a clear negative societal impact, however the problem of unfair algorithmic design is pervasive.

Unfortunately, these are problems that cannot be solved by a single algorithm, or even a single field of research, nor should they. Resolution will require careful interdisciplinary work to define practical notions of fairness and develop algorithms that achieve these notions. To contribute to these efforts, I study partitioning algorithms that exhibit representational equality. In addition, I co-led a AAAI 2022 tutorial on fair clustering (`https://www.fairclustering.com/`). This tutorial creates a clear taxonomy of the various notions of fairness that have been considered in research, overviews results and methods across fair clustering, and provides suggested practices and cautions for the use of fairness in real systems.

Here, we study the problem of fair hierarchical clustering. "Clustering" means partitioning data into "clusters" without any provided data structure (i.e., it is unsupervised). Hierarchical clustering extends this notion to form a tiered structure of clusters containing clusters.

This work consisted of two projects: 1) initiating the study of fair hierarchical clustering [Ahmadian et al., 2020b] (alongside the concurrent work of [Chhabra and Mohapatra, 2022],

which did not consider rigorous guarantees as we did) and 2) massive improvements of our results in terms of practical use, theoretical guarantees, and explainability [Knittel et al., 2023a, Knittel et al., 2023b]. Generally speaking, fair hierarchical clustering can be leveraged for a number of applications. For instance, news articles can be partitioned into a topic hierarchy and we must ensure that viewpoints are not over-represented in a given topic. Geographic regions, too, have natural hierarchical partitions, and we may need to consider the representation of protected features (i.e., race or gender) of individuals in these regions. More broadly, consider any application of clustering where fairness is mandatory but the number of desired clusters is not known. A fair hierarchical clustering can then simultaneously yield clusterings at a number of different resolutions that "agree with each other"[1]. These clusterings can be evaluated and selected independently depending on the application.

Our inputs are complete graphs, where edge weights can denote data *similarity* or *difference*, depending on the context. A *fairness constraint* defines a criteria to determine if a single cluster is fair. A hierarchical clustering is *fair* if each cluster is fair. In our work, we utilize [Dasgupta, 2016]'s objective, *cost*, and two related functions, *revenue* [Moseley and Wang, 2017] and *value* [Cohen-Addad et al., 2018], to measure the quality of our output. Our results apply to a broad family of fairness constraints that are *union-closed*, meaning that if two clusters are considered fair, then the union of the two clusters is also considered fair. Notably, our results hold for the most general notion of bounded representational fairness [Bera et al., 2019]: given a color assignment to all vertices and vectors $\vec{\alpha}$ and $\vec{\beta}$ such that $0 \leq \alpha_i \leq \beta_i \leq 1$ for any color i, a cluster is fair if color i represents between an α_i and β_i fraction of its vertices.

[1] If two vertices are in the same high-resolution cluster, then they will be in the same low-resolution cluster. This means the clusterings represent the data in similar ways.

Our results cover all three objectives. For revenue and value, our initial work [Ahmadian et al., 2020b] proposes nice extensions of state-of-the-art approximation algorithms to the fairness setting. Our methods are an application of [Chierichetti et al., 2017]'s fairlet decomposition method, which illustrates the power of this technique. We show that, with small modifications, these algorithms achieve the same asymptotic approximations of the objectives in many settings (i.e., many different color proportions in the overall data) and they empirically exhibit little to no loss in clustering quality. Ultimately, these algorithms are a nice, simple, and practical extension of popular existing algorithms in this field.

Cost proved to be a much more difficult objective. We initially designed an entirely novel algorithm that achieves an $O(n^{5/6}\log^{5/4} n)$-approximation of cost [Ahmadian et al., 2020b] in the two color setting. While this was not a practical solution, it was an entirely innovative approach that showed that, even with careful algorithmic analysis that balances trade-offs between a number of opposing parameters, it is still intuitively much harder to achieve strong approximation guarantees under cost than under revenue and value. This generally agrees with work in (unfair) hierarchical clustering, where revenue and value both yield small, constant-factor approximations [Alon et al., 2020, Cohen-Addad et al., 2018], but cost is not even constant-factor approximable under a widely believed conjecture called the *Small Set Expansion Hypothesis*, and the best known approximation is $O(\sqrt{\log n})$ [Charikar and Chatziafratis, 2017].

In a second [Knittel et al., 2023b] and third work [Knittel et al., 2023a], I made great strides to improve the approximation factor achieved for fair low-cost hierarchical clustering, the former achieving a near-polylogarithmic approximation, $O(n^{\delta}\text{polylog}\,(n))$ for arbitrarily small $\delta = O(1)$, and the latter achieving a true polylogarithmic approximation. They both guarantee that the fraction of any color in any cluster deviates from the true proportion of that color by at

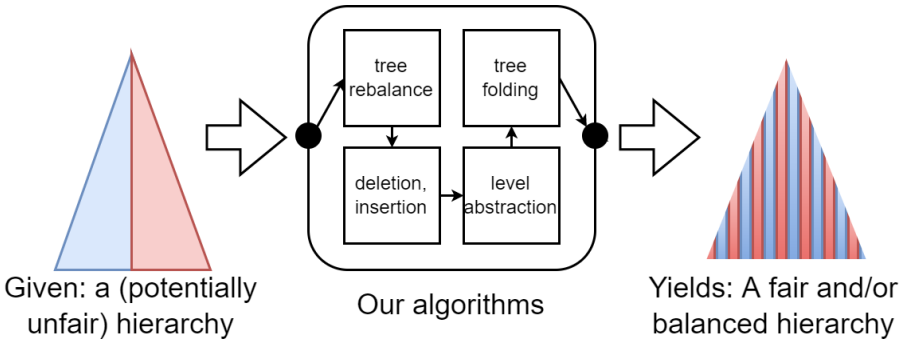


Figure 1.1: Our algorithms take a potentially unfair hierarchical clustering, apply our tree operators, and yield fair and/or balanced hierarchies.

most a factor of $O(\exp(1/\delta))$. In other words, they are parameterizable by $\delta \in (0, 1)$, which determines a trade-off between approximation factor and degree of unfairness. On a given low-cost hierarchical clustering, both algorithms apply local tree operations to rebalance the tree and fairly distribute colors (see Figure 1.1). Not only are these operations easy to implement, but they nicely quantify exactly how the hierarchy is being altered. This simplifies our analysis and makes the algorithms explainable, an issue not yet commonly studied in hierarchical clustering to our knowledge.

1.2 Massively Parallel Algorithms

Our second question largely concerns the increasing scale of data alongside the growing demand for faster products. As data growth begins to outpace hardware speedup in this post-Moore's Law era, efficient algorithmic design is necessary to ensure perform at scale. Massively parallel computing on distributed systems, where individual machines work on small subsets of the input in parallel, is a natural solution. Not only does it leverage the compute power of parallelism, but it also realistically models commodity hardware by assuming individual machines *cannot* store the entire input. To this end, in the interest of addressing our first question, I study

massively parallel algorithms on graphs.

My work leverages Karloff et al.'s [Karloff et al., 2010] *Massively Parallel Computation* (MPC) model to develop large-scale graph algorithms. MPC is at the forefront of computation at scale, and it has been extensively studied and applied to a vast range of graph problems in recent years. The MPC design models the behavior of programming paradigms such as MapReduce [Dean and Ghemawat, 2008] and frameworks like Flume [Chambers et al., 2010], Hadoop [White, 2009], and Spark [Zaharia et al., 2010]. Therefore, theoretical progress in MPC has great potential for real impact on modern technological systems.

MPC is most often used to solve problems on graphs with n vertices and m edges. In the *linear regime*, individual machines are only able to store up to $\widetilde{O}(n)$[2] bits in their memory. In the *sublinear regime*, this reduces to $O(n^{\epsilon})$ bits where $0 < \epsilon < 1$. Both of these regimes are considered highly efficient and can be practical in a number of applications, however the sublinear regime is the gold standard.

In MPC, algorithms start with an arbitrary distribution of data across a number of these space-constrained machines. The algorithm then proceeds in rounds. In each round, a machine executes a local polynomial time computation on its input. At the end of each round, machines may send and receive messages within their memory constraints. This continues until the algorithm completes. MPC algorithms are considered highly scalable if they require only logarithmic rounds. Some more ambitious algorithms manage to achieve complexities like $O(\sqrt{\log n})$ (notably, maximum matching) and $O(\log \log n)$. The gold standard here, of course, is the constant round algorithm that may run in $O(1)$ or $O(\mathsf{poly}(\epsilon))$ rounds. The total space used by the algorithm must be bounded by $\widetilde{O}(n + m)$.

[2]We say $\widetilde{O}(f(n)) = O(f(n) \log n)$.

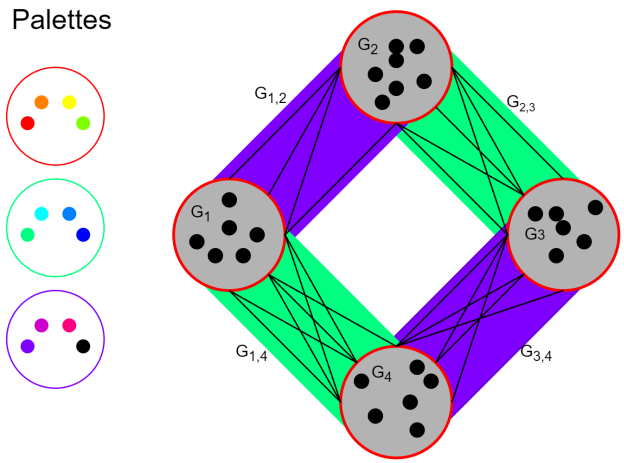


Figure 1.2: The color assignment process, assuming four partitions (note: graphs $G_{1,3}$ and $G_{2,4}$ are omitted for simplicity). We require three palettes. The first is used to color edges in G_1, G_2, G_3, G_4, the second for $G_{2,3}$ and $G_{1,4}$, and the last for $G_{1,2}$ and $G_{3,4}$ (a fourth would be required to color $G_{1,3}$ and $G_{2,4}$). No two edges being colored using the same palette shares an end point.

Much of the research in MPC is dedicated to solving classical graph theoretic problems such as graph matching, coloring, and independent set. Many algorithms that solve these problems have a vast range of applications and often run as subroutines in nearly all technologies we use today. My research in MPC has led to new, inventive solutions for geometric data embeddings [Ahanchi et al., 2023], hierarchical clustering [Hajiaghayi and Knittel, 2020], and edge coloring [Behnezhad et al., 2019b] that reduce necessary communication and memory. Ultimately, I strive to develop novel MPC algorithms that enable technologies to meet or exceed scalability demands as datasets continue to grow.

1.2.1 Edge Coloring

Inspired by the recent work of [Chang et al., 2018] which found near-optimal bounds for MPC vertex coloring and effectively closed the problem, we considered the next natural problem: edge coloring. Edge coloring is the problem of assigning colors to edges in a graph such that no

two adjacent edges share a color. We want to minimize the number of colors required to do this. We [Behnezhad et al., 2019b] start by random allocating vertices to k partitions, $V_1, \ldots, V_k$. With high probability, the max degree Δ_i in each V_i is bounded. We then perform a local edge coloring using Δ_i colors on each V_i as guaranteed by Vizing's [Vizing, 1964] celebtrated theorem. Since any edge in V_i is not adjacent to any edge in V_j, we can assign the same palette (or, set of colors) to each partition. Additionally, for any two V_i and V_j, let E_{ij} be the set of edges with one endpoint in each. We can again find a Δ-coloring locally on this graph, and we can again show that for many E_{ij} and $E_{k\ell}$, we can reuse the same palette of colors. Ultimately, we achieve a $\left(\Delta + \widetilde{O}\left(\Delta^{3/4}\right)\right)$-coloring in $\widetilde{O}(n)$ machine space and $O(1)$ rounds, outperforming previous work.

1.2.2 Hierarchical Clustering

This work concerns the problem of *scalable* hierarchical clustering, but this time *without* fairness constraints. In [Hajiaghayi and Knittel, 2020], I introduced Matching Affinity Clustering (MAC), the successor to [Bateni et al., 2017]'s MPC hierarchical clustering algorithm, Affinity Clustering (AC). AC constructs a hierarchy based off the minimum spanning tree. It exhibits state-of-the-art empirical performance and is used in real products, such as Google's balanced partitioning algorithms. Unfortunately, it lacks strong theoretical guarantees and can create arbitrarily unbalanced hierarchies. Inspired by AC's use of graph theoretic structures, MAC leverages scalable matching algorithms to iteratively match and join clusters. This guarantees near perfect balance and we can show that MAC provides a good approximation for both revenue and value. MAC is the first hierarchical clustering algorithm to simultaneously achieve: broad theoretical

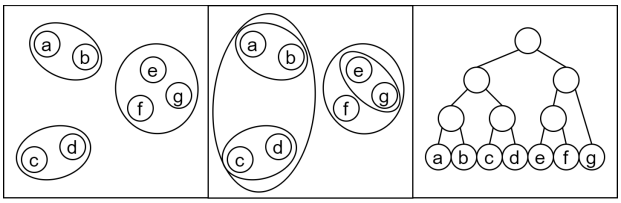


Figure 1.3: On the left is a 3-clustering, in the center is a hierarchical clustering, and on the right is its dendrogram.

guarantees, strong empirical success, cluster balance, and scalability.

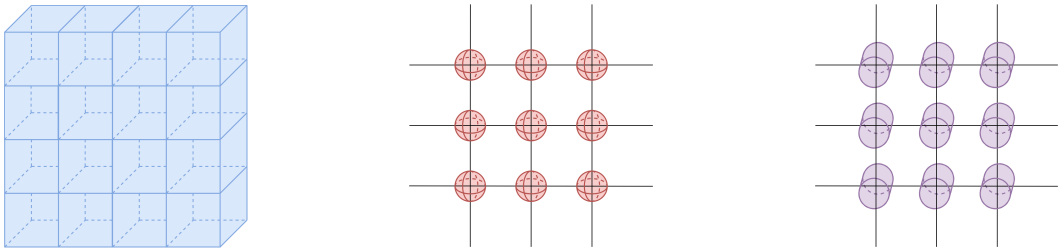
1.2.3 Tree Embeddings in High Dimensions

Our final MPC work [Ahanchi et al., 2023] addresses tree embeddings of high-dimensional data. Often, massive datasets are represented by extremely high-dimensional vectors, which can be highly impractical to work with. Therefore, there is much interest in dimension-reducing data embeddings that preserve the relational meaning between data (i.e., in our case, the distance between datapoints). In this work, we use MPC to efficiently transform high-dimensional data into tree embeddings, where data is stored on a tree of size $O(n)$, and the distance between data is encoded by the length of the shortest path between vertices in the tree.

The main strategy here is to use a data partitioning scheme to construct a *hierarchically well-separated tree* (HST)[3] of the data. The HST will be our final embedding. We consider two famous geometric data partitioning methods: [Arora, 1997]'s grid partitioning and [Charikar et al., 1998]'s ball partitioning. In grid partitioning, we create an even grid of cell width w over the dataset where the origin is shifted by a uniform random vector $\vec{r}$. Each cell forms a cluster (a node in our HST) of the points within the cell, and we recurse with a random shifted grid of smaller cell width $w' < w$, stopping when a cell contains at most one point. These cells will

[3] An HST is a weighted tree whose leaves are the input data. The distance between input data is determined by the tree metric defined by the HST.

be the children of their parent cluster in the HST, and the leaves will represent individual input points. On the other hand, in ball partitioning, we use a random shifted grid to define the centers of balls. We place a ball of radius $w/4$ centered at each grid point, and use these to cluster the data. Obviously, this does not encompass the entire space, and this must be repeated some number of times. We then recurse in the same way and the HST is constructed similarly.



(a) Grid partitioning, $w = 1$. (b) Ball partitioning, $w = 1/4$. (c) Hybrid partitioning, $w = 1/4$.

Figure 1.4: We depict one level (and one sample) of each discussed partitioning method on 3-dimensional space. In grid partitioning (12.1a), we partition the grid into hypercubic cells of width 1 shifted by a random vector. In ball partitioning (12.1b), we place a ball of radius $1/4$ at each intersection of grid boundaries. In hybrid partitioning (12.1c) with $r = 2$, we run a ball partitioning with ball radius $1/4$ on buckets of dimensions. In this example, we bucket $\{x, y\}$ together (hence, the circular xy cross-sections) and $\{z\}$ separately.

Due to the number of grids required, ball partitioning cannot be implemented in MPC's space requirements. While grid partitioning can, it does not provide an acceptable distortion (i.e., proportional deviation between true and embedded distance). To combat this, we propose a *hybrid grid and ball partitioning algorithm*, where we create buckets of subsets of the dimensions of the data. For each bucket, we do a ball partition on the data projected into the dimensions of the bucket. For instance, on 3 dimensions with axes x, y, and z, the bucket of dimensions $\{1, 2\}$ corresponds to projecting data into the xy plane and then running a ball partitioning. In the end, these methods require $O((nd)^\epsilon)$ space (for dimensionality d) and run highly efficiently in $O(1)$ rounds. They produce tree embeddings with $\widetilde{O}(\log^{1.5} n)$ distortion and imply results for Earth mover distance, minimum spanning tree, maximum cut, and densest ball.

1.3 Adaptive Massively Parallel Computation

In order to keep up with the increasing pace of big data, we must be constantly critical of our tools and models so that we may identify when a tool is insufficient to solve a problem. On that note, one of the most inhibiting qualities of the MPC model is the widely believed 1vs2Cycle conjecture. This posits that distinguishing between an n-lengthed cycle or two disjoint $n/2$-lengthed cycles requires $O(\log n)$ rounds in MPC. This has significant implications on a wide range of other graph problems, yielding lower bounds for connectivity [Behnezhad et al., 2019d], matching [Ghaffari et al., 2019a, Nanongkai and Scquizzato, 2019], clustering [Yaroslavtsev and Vadapalli, 2018], and more [Andoni et al., 2019, Ghaffari et al., 2019a, Lacki et al., 2020]. To combat these lower bounds, a novel extension of the MPC model, *Adaptive Massively Parallel Computing* (AMPC), was introduced by Behnezhad et al. [Behnezhad et al., 2019c]. AMPC improves upon MPC by adding the power of a *distributed hash table*, which allows individual machines to select their data over the course of a round, as opposed to being sent the data before the round starts. Specifically, machines are given access to a sequence of distributed hash tables, $\{\mathcal{H}_i\}_{i=1,2,\ldots}$, which are shared data structures of size $\widetilde{O}(n+m)$. In the ith round, all machines are allowed in-round read-only access to $\mathcal{H}_i$ and write-only access to $\mathcal{H}_{i+1}$.

Predating the formal notion of AMPC, MPC with a distributed hash table had already been studied in the context of finding connected components [Kiveris et al., 2014] and hierarchical clusterings [Bateni et al., 2017], expressing its power and also its practicality in real world systems. Since its conception, a number of works have exhibited both the power [Behnezhad et al., 2019c, Behnezhad et al., 2020] and limitations [Charikar et al., 2020] of the AMPC model. Notably, the 1vs2Cycle conjecture does not hold in AMPC, and thus AMPC is a promising tool

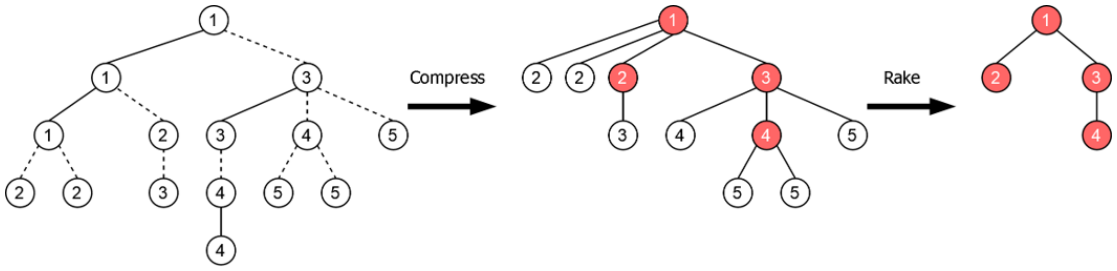


Figure 1.5: An illustration of our algorithm's compress (where contiguous groups of vertices in a pre-order traversal are contracted) and rake operations on low degree trees. On high degree trees, the same compress and rake framework is followed, but the low-degree algorithm is used as the compress function.

to further scalable algorithmic performance. My recent work in this area addresses tree contractions (a general technique for algorithms on trees) and minimum cut computation. As this is a new model, I am one of the first researchers to study AMPC.

1.3.1 Tree Contraction

This work [Hajiaghayi et al., 2022b] proposes an AMPC adaptation of Miller and Reif's [Miller and Reif, 1985] tree contraction. Initially implemented in PRAM, or parallel RAM, standard tree contraction provides a general tool for solving a wide range of problems on trees. This algorithm involves a two step process: (1) compress, where every other vertex on a long chain of vertices with degree two is contracted into its parent and (2) rake, where all leaves are contracted into their parent. It is not hard to see that this results in an $O(\log n)$-time PRAM algorithm to contract a tree into a single vertex. This $O(\log n)$ barrier has not been broken since the algorithm's inception 25 years ago. We are the first to show that there exists a model, AMPC, that can implement highly efficient constant-round tree contractions.

In order to do this, we first consider a pre-order traversal of the tree. This creates an ordering over the vertices. We create a "group" of vertices for every contiguous chunk of about n^ϵ vertices

in the ordering (where ϵ is our AMPC memory limit parameter). To compress, we contract each connected component within each group into a single vertex. To rake, for every set of leaves who share a parent, we iteratively contract them into each other in groups of n^ϵ until they fit on one machine, at which point we contract them into their parent vertex. We then rigorously show that this reduces the graph size by a factor of n^ϵ, thus requiring only $O(1/\epsilon)$ rounds. This, however, only works on trees of max degree n^ϵ. On general trees, we simply group vertices into low-degree components, and then use the low-degree algorithm to compress each component. This defines the compress stage, and the rake stage is effectively the same. Ultimately, this defines an $O(1/\epsilon^3)$-round sublinear AMPC algorithm for tree contractions, greatly exceeding the efficiency of its PRAM counterpart.

In addition, we show a generalized technique for solving problems in constant sublinear AMPC rounds using this framework. The most natural such problems are dynamic programs on trees that can be solved in a bottom-up fashion and admit *connected and sibling contracting functions*. A connected contracting function takes in a connected component of size up to $O(n^\epsilon)$, contracts it into a single vertex, and assigns the new vertex v data D_v such that given a solution to v's children, D_v can be used to compute the dynamic program value at the component's root. We require D_v to fit in $\widetilde{O}(\deg(v))$ space, where $\deg(v)$ is v's degree after contraction. Sibling contracting functions are similar, but they contract a parent and its leaf children. They must allow for efficient AMPC computation on parents with many children (i.e., exceeding the $O(n^\epsilon)$ local space constraint). We show that the dynamic programs required to solve weighted tree matching, weighted tree independent set, and tree isomorphism satisfy these, and thus can be solved in constant AMPC rounds. We expect, however, that our algorithm is much more widely applicable.

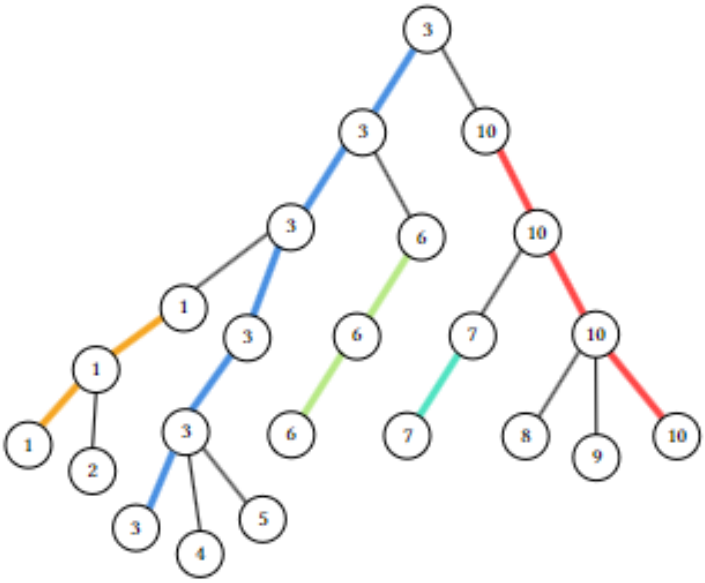


Figure 1.6: A heavy light decomposition. Heavy paths are denote in bolded color. These paths are replaced by complete binary trees with the path in the leaves. The tree becomes low-depth, and we can extract a labeling.

1.3.2 Minimum Cut

In the minimum cut problem, we are asked to partition the graph into two sets such that the weight of the edges crossing from one set to another is minimized. The current best MPC algorithm by Ghaffari and Nowicki [Ghaffari and Nowicki, 2020] requires $O(\log n \log\log n)$ rounds. In their work, they show that by contracting a graph along the edges of the minimum spanning tree on random-weighted edges many times in parallel, it is likely that at some point in one of the parallel processes, the minimum singleton cut (i.e., one that isolates a single vertex) is an approximate minimum cut in the expanded graph.

Using their general method, we create an entirely different underlying implementation for decomposing the graph and computing the minimum singleton cut at each step. We start by executing a low depth tree decomposition, which assigns a labeling to vertices and allows for extremely fast parallel decomposition of a given tree. This can be created efficiently in AMPC

by leveraging its ability to root and orient trees as well as identify connected components to restructure long paths in this tree with complete binary trees. This restructuring reduces the depth of the tree and allows for fast vertex labeling according to vertex depth in this new tree. After obtaining our labeling, we can then leverage the implied decomposition and a counting argument on the singleton cuts in order to track each singleton cut without executing too much redundant computation. In the end, this cuts down an $O(\log n)$-round MPC computation from Ghaffari and Nowicki to take $O(1)$ rounds in AMPC. What we are left with is a complete $O(\log \log n)$-round AMPC algorithm that solves minimum cut in the sublinear regime, an exponential round complexity speedup over the work of Ghaffari and Nowicki.

1.4 Roadmap

This dissertation covers the aforementioned works in their entirety. It is divided into three parts. In Part I, we discuss our results in fair hierarchical clustering. Chapter 2 discusses our first work which initiated the field of research, and Chapters 3 and 4 describe the follow-up works. In Part II, we turn our attention to scalability through the lens of Massively Parallel Computation. Here, we present our work on edge coloring (Chapter 10), hierarchical clustering (Chapter 11), and tree embeddings (Chapter 12). Finally, we end our discussion on scalable graph algorithms with our Adaptive MPC works on tree contraction in Chapter 15 and minimum cut in Chapter 16.

Part I

Fair Hierarchical Clustering

Chapter 2: Fair Hierarchical Clustering

2.1 Introduction

Algorithms and machine learned models are increasingly used to assist in decision making on a wide range of issues, from mortgage approval to court sentencing recommendations [Kleinberg et al., 2017a]. It is clearly undesirable, and in many cases illegal, for models to be biased to groups, for instance to discriminate on the basis of race or religion. Ensuring that there is no bias is not as easy as removing these protected categories from the data. Even without them being explicitly listed, the correlation between sensitive features and the rest of the training data may still cause the algorithm to be biased. This has led to an emergent literature on computing provably fair outcomes (see the book [Barocas et al., 2019]).

The prominence of clustering in data analysis, combined with its use for data segmentation, feature engineering, and visualization makes it critical that efficient fair clustering methods are developed. There has been a flurry of recent results in the ML research community, proposing algorithms for fair *flat* clustering, i.e., partitioning a dataset into a set of disjoint clusters, as captured by K-CENTER, K-MEDIAN, K-MEANS, correlation clustering objectives [Ahmadian et al., 2019, Ahmadian et al., 2020c, Backurs et al., 2019, Bera et al., 2019, Bercea et al., 2019, Chen et al., 2019, Chiplunkar et al., 2020, Huang et al., 2019, Jones et al., 2020, Kleindessner et al., 2019a, Karloff et al., 2010]. However, the same issues affect *hierarchical clustering*, which is the

problem we study.

The input to the hierarchical clustering problem is a set of data points, with pairwise similarity or dissimilarity scores. A hierarchical clustering is a tree, whose leaves correspond to the individual datapoints. Each internal node represents a cluster containing all the points in the leaves of its subtree. Naturally, the cluster at an internal node is the union of the clusters given by its children. Hierarchical clustering is widely used in data analysis [Dubes and Jain, 1980], social networks [Mann et al., 2008,Rajaraman and Ullman, 2011], and image/text organization [Karypis et al., 2000].

Hierarchical clustering is frequently used for flat clustering when the number of clusters is a priori unknown. A hierarchical clustering yields a set of clusterings at different granularities that are consistent with each other. Therefore, in all clustering problems where fairness is desired but the number of clusters is unknown, fair hierarchical clustering is useful. As concrete examples, consider a set of news articles organized by a topic hierarchy, where we wish to ensure that no single source or view point is over-represented in a cluster; or a hierarchical division of a geographic area, where the sensitive attribute is gender or race, and we wish to ensure balance in every level of the hierarchy. There are many such problems that benefit from fair hierarchical clustering, motivating its study.

Our contributions We initiate an algorithmic study of fair hierarchical clustering. We build on Dasgupta's seminal formal treatment of hierarchical clustering [Dasgupta, 2016] and prove our results for the revenue [Moseley and Wang, 2017], value [Cohen-Addad et al., 2018], and cost [Dasgupta, 2016] objectives in his framework.

To achieve fairness, we show how to extend the fairlets machinery, introduced by [Chierichetti

et al., 2017] and extended by [Ahmadian et al., 2019], to this problem. We then investigate the complexity of finding a good fairlet decomposition, giving both strong computational lower bounds and polynomial time approximation algorithms.

Finally, we conclude with an empirical evaluation of our approach. We show that ignoring protected attributes when performing hierarchical clustering can lead to unfair clusters. On the other hand, adopting the fairlet framework in conjunction with the approximation algorithms we propose yields *fair* clusters with a *negligible* objective degradation.

Related work Hierarchical clustering has received increased attention over the past few years. Dasgupta [Dasgupta, 2016] developed a cost function objective for data sets with similarity scores, where similar points are encouraged to be clustered together lower in the tree. Cohen-Addad et al. [Cohen-Addad et al., 2018] generalized these results into a class of optimization functions that possess other desirable properties and introduced their own value objective in the dissimilarity score context. In addition to validating their objective on inputs with known ground truth, they gave a theoretical justification for the average-linkage algorithm, one of the most popular algorithms used in practice, as a constant-factor approximation for value. Contemporaneously, Moseley and Wang [Moseley and Wang, 2017] designed a revenue objective function based on the work of Dasgupta for point sets with similarity scores and showed the average-linkage algorithm is a constant approximation for this objective as well. This work was further improved by Charikar [Charikar et al., 2019a] who gave a tighter analysis of average-linkage for Euclidean data for this objective and [Ahmadian et al., 2020a, Alon et al., 2020] who improved the approximation ratio in the general case.

In parallel to the new developments in algorithms for hierarchical clustering, there has been

tremendous development in the area of fair machine learning. We refer the reader to a recent textbook [Barocas et al., 2019] for a rich overview, and focus here on progress for fair clustering. Chierichetti et al. [Chierichetti et al., 2017] first defined fairness for k-median and k-center clustering, and introduced the notion of *fairlets* to design efficient algorithms. Extensive research has focused on two topics: adapting the definition of fairness to broader contexts, and designing efficient algorithms for finding good fairlet decompositions. For the first topic, the fairness definition was extended to multiple values for the protected feature [Ahmadian et al., 2019, Bercea et al., 2019, Rösner and Schmidt, 2018]. For the second topic, Backurs et al. [Backurs et al., 2019] proposed a near-linear constant approximation algorithm for finding fairlets for k-median, Schmidt et al. [Schmidt et al., 2019] introduced a streaming algorithm for scalable computation of coresets for fair clustering, Kleindessner et al. [Kleindessner et al., 2019a] designed a linear time constant approximation algorithm for k-center where cluster centers are selected proportionally from a set of colors, Bercea et al. [Bercea et al., 2019] developed methods for fair k-means, while Ahmadian et al. [Ahmadian et al., 2020c] and Ahmadi et al. [Ahmadi et al., 2020] defined approximation algorithms for fair correlation clustering. Concurrently with our work, Chhabra et al. [Chhabra and Mohapatra, 2022] introduced a possible approach to ensuring fairness in hierarchical clustering. However, their fairness definition differs from ours (in particular, they do not ensure that all levels of the tree are fair), and the methods they introduce are heuristic, without formal fairness or quality guarantees.

Beyond clustering, the same balance notion that we use has been utilized to capture fairness in other contexts, for instance: fair voting [Celis et al., 2018], fair optimization [Chierichetti et al., 2019], as well as other problems.

2.2 Formulation

2.2.1 Objectives for hierarchical clustering

Let $G = (V, s)$ be an input instance, where V is a set of n data points, and $s : V^2 \to \mathbb{R}^{\geq 0}$ is a similarity function over vertex pairs. For two sets, $A, B \subseteq V$, we let $s(A, B) = \sum_{a \in A, b \in B} s(a, b)$ and $S(A) = \sum_{\{i,j\} \in A^2} s(i, j)$. For problems where the input is $G = (V, d)$, with d a distance function, we define $d(A, B)$ and $d(A)$ similarly. We also consider the *vertex-weighted* versions of the problem, i.e. $G = (V, s, m)$ (or $G = (V, d, m)$), where $m : V \to \mathbb{Z}^+$ is a weight function on the vertices. The vertex-unweighted version can be interpreted as setting $m(i) = 1, \forall i \in V$. For $U \subseteq V$, we use the notation $m(U) = \sum_{i \in U} m(i)$.

A *hierarchical clustering* of G is a tree whose leaves correspond to V and whose internal vertices represent the merging of vertices (or clusters) into larger clusters until all data merges at the root. The goal of hierarchical clustering is to build a tree to optimize some objective.

To define these objectives formally, we need some notation. Let T be a hierarchical clustering tree of G. For two leaves i and j, we say $i \vee j$ is their least common ancestor. For an internal vertex u in T, let $T[u]$ be the subtree in T rooted at u. Let $\text{leaves}(T[u])$ be the leaves of $T[u]$.

We consider three different objectives—*revenue*, *value*, and *cost*—based on the seminal framework of [Dasgupta, 2016], and generalize them to the vertex-weighted case.

Revenue. Moseley and Wang [Moseley and Wang, 2017] introduced the revenue objective for hierarchical clustering. Here the input instance is of the form $G = (V, s, m)$, where $s : V^2 \to \mathbb{R}^{\geq 0}$ is a *similarity* function.

Definition 1 (Revenue)**.** *The* revenue *(*rev*) of a tree* T *for an instance* $G = (V, s, m)$*, where* $s(\cdot, \cdot)$

denotes similarity between data points, is: $\text{rev}_G(T) = \sum_{i,j \in V} s(i,j) \cdot \big(m(V) - m(\text{leaves}(T[i \vee j]))\big)$.

Note that in this definition, each weight is scaled by (the vertex-weight of) the non-leaves. The goal is to find a tree of maximum revenue. It is known that average-linkage is a $1/3$-approximation for vertex-unweighted revenue [Moseley and Wang, 2017]; the state-of-the-art is a 0.585-approximation [Alon et al., 2020].

As part of the analysis, there is an upper bound for the revenue objective [Cohen-Addad et al., 2018, Moseley and Wang, 2017], which is easily extended to the vertex-weighted setting:

$$\text{rev}_G(T) \leq \Big(m(V) - \min_{u,v \in V, u \neq v} m(\{u,v\})\Big) \cdot s(V). \tag{2.1}$$

Note that in the vertex-unweighted case, the upper bound is just $(|V| - 2)s(V)$.

Value. A different objective was proposed by Cohen-Addad et al. [Cohen-Addad et al., 2018], using distances instead of similarities. Let $G = (V, d, m)$, where $d : V^2 \to \mathbb{R}^{\geq 0}$ is a distance (or dissimilarity) function.

Definition 2 (Value)**.** *The* value *(*val*) of a tree T for an instance $G = (V, d, m)$ where $d(\cdot, \cdot)$ denotes distance is:* $\text{val}_G(T) = \sum_{i,j \in V} d(i,j) \cdot m(\text{leaves}(T[i \vee j]))$.

As in revenue, we aim to find a hierarchical clustering to maximize value. Cohen-Addad et al. [Cohen-Addad et al., 2018] showed that both average-linkage and a locally ϵ-densest cut algorithm achieve a $2/3$-approximation for vertex-unweighted value. They also provided an upper bound for value, much like that in (2.1), which in the vertex-weighted context, is:

$$\text{val}_G(T) \leq m(V) \cdot d(V). \tag{2.2}$$

Cost. The original objective introduced by Dasgupta [Dasgupta, 2016] for analyzing hierarchical clustering algorithms introduces the notion of cost.

Definition 3 (Cost)**.** *The* cost *of a tree T for an instance $G = (V, s)$ where $s(\cdot, \cdot)$ denotes similarity is:* $\text{cost}_G(T) = \sum_{i,j \in V} s(i, j) \cdot |\,\text{leaves}(T[i \vee j])|$.

The objective is to find a tree of minimum cost. From a complexity point of view, cost is a harder objective to optimize. Charikar and Chatziafratis [Charikar and Chatziafratis, 2017] showed that cost is not constant-factor approximable under the Small Set Expansion hypothesis, and the current best approximations are $O\left(\sqrt{\log n}\right)$ and require solving SDPs.

Convention. Throughout the paper we adopt the following convention: $s(\cdot, \cdot)$ will always denote similarities and $d(\cdot, \cdot)$ will always denote distances. Thus, the inputs for the cost and revenue objectives will be instances of the form (V, s, m) and inputs for the value objective will be instances of the form (V, d, m). All the missing proofs can be found in the Supplementary Material.

2.2.2 Notions of fairness

Many definitions have been proposed for fairness in clustering. We consider the setting in which each data point in V has a *color*; the color corresponds to the protected attribute.

Disparate impact. This notion is used to capture the fact that decisions (i.e., clusterings) should not be overly favorable to one group versus another. This notion was formalized by Chierichetti et al. [Chierichetti et al., 2017] for clustering when the protected attribute can take on one of two values, i.e., points have one of two colors. In their setup, the *balance* of a cluster is the ratio of the minimum to the maximum number of points of any color in the cluster. Given

a balance requirement t, a clustering is fair if and only if each cluster has a balance of at least t.

Bounded representation. A generalization of disparate impact, bounded representation focuses on mitigating the imbalance of the representation of protected classes (i.e., colors) in clusters and was defined by Ahmadian et al. [Ahmadian et al., 2019]. Given an over-representation parameter α, a cluster is fair if the *fractional representation* of each color in the cluster is at most α, and a clustering is fair if each cluster has this property. This was further generalized by Bera et al. [Bera et al., 2019] and Bercea et al. [Bercea et al., 2019]. They introduce vectors $\vec{\alpha}, \vec{\beta}$ such that for a cluster to be fair, for each color c_i, the fractional representation of c_i in the cluster must be between β_i and α_i. We discuss our results in terms of the over-representation constraint by [Ahmadian et al., 2019], however many of these results extend to this more general setting given an appropriate fairlet decomposition. An interesting special case of this notion is when there are c total colors and $\alpha = 1/c$. In this case, we require that every color is equally represented in every cluster. We will refer to this as *equal representation*. These notions enjoy the following useful property:

Definition 4 (Union-closed)**.** *A fairness constraint is* union-closed *if for any pair of fair clusters A and B, $A \cup B$ is also fair.*

This property is useful in hierarchical clustering: given a tree T and internal node u, if each child cluster of u is fair, then u must also be a fair cluster.

Definition 5 (Fair hierarchical clustering)**.** *For any fairness constraint, a* hierarchical clustering is fair *if all of its clusters (besides the leaves) are fair.*[1]

[1]According to the definition, a hierarchical clustering tree might be fair even if every layer (apart from the root) is an unfair clustering. For example, consider a tree that splits off one singleton at its root. Every layer in the tree apart from the root will contain this singleton and thus is an unfair clustering. An alternative way of defining a fair tree is to enforce that the tree has to contain a layer of fairlets of some small size. The results of this paper extend to either definition.

Thus, under any union-closed fairness constraint, this definition is equivalent to restricting the bottom-most clustering (besides the leaves) to be fair. Then given an objective (e.g., revenue), the goal is to find a fair hierarchical clustering that optimizes the objective. We focus on the bounded representation fairness notion with c colors and an over-representation cap α. However, the main ideas for the revenue and value objectives work under any notion of fairness that is union-closed.

2.3 Fairlet decomposition

Definition 6 (Fairlet [Chierichetti et al., 2017])**.** *A* fairlet Y *is a fair set of points such that there is no partition of* Y *into* Y_1 *and* Y_2 *with both* Y_1 *and* Y_2 *being fair.*

In the bounded representation fairness setting, a set of points is fair if at most an α fraction of the points have the same color. We call this an α-*capped fairlet*. For $\alpha = 1/t$ with t an integer, the fairlet size will always be at most $2t - 1$. We will refer to the maximum size of a fairlet by m_f.

Recall that given a union-closed fairness constraint, if the bottom clustering in the tree is a layer of fairlets (which we call a *fairlet decomposition* of the original dataset) the hierarchical clustering tree is also fair. This observation gives an immediate algorithm for finding fair hierarchical clustering trees in a two-phase manner. (i) Find a fairlet decomposition, i.e., partition the input set V into clusters $Y_1, Y_2, \ldots$ that are all fairlets. (ii) Build a tree on top of all the fairlets. Our goal is to complete both phases in such a way that we optimize the given objective (i.e., revenue or value).

In Section 2.4, we will see that to optimize for the revenue objective, all we need is a fairlet

decomposition with bounded fairlet size. However, the fairlet decomposition required for the value objective is more nuanced. We describe this next.

Fairlet decomposition for the value objective For the value objective, we need the total distance between pairs of points inside each fairlet to be small. Formally, suppose V is partitioned into fairlets $\mathcal{Y} = \{Y_1, Y_2, \ldots\}$ such that Y_i is an α-capped fairlet. The cost of this decomposition is defined as:

$$\phi(\mathcal{Y}) = \sum_{Y \in \mathcal{Y}} \sum_{\{u,v\} \subseteq Y} d(u, v). \tag{2.3}$$

Unfortunately, the problem of finding a fairlet decomposition to minimize $\phi(\cdot)$ does not admit any constant-factor approximation unless P = NP.

Theorem 7. *Let $z \geq 3$ be an integer. Then there is no bounded approximation algorithm for finding $(\frac{z}{z+1})$-capped fairlets optimizing $\phi(\mathcal{Y})$, which runs in polynomial time, unless P = NP.*

The proof proceeds by a reduction from the Triangle Partition problem, which asks if a graph $G = (V, E)$ on $3n$ vertices can be partitioned into three element sets, with each set forming a triangle in G. Fortunately, for the purpose of optimizing the value objective, it is not necessary to find an approximate decomposition.

2.4 Optimizing revenue with fairness

This section considers the revenue objective. We will obtain an approximation algorithm for this objective in three steps: (i) obtain a fairlet decomposition such that the maximum fairlet size in the decomposition is small, (ii) show that any β-approximation algorithm to (2.1) (i.e., any algorithm that achieves a β-factor approximation of (2.1) for some given β) plus this

fairlet decomposition can be used to obtain a (roughly) β-approximation for fair hierarchical clustering under the revenue objective, and (iii) use average-linkage, which is known to be a $1/3$-approximation to (2.1). (We note that the recent work [Ahmadian et al., 2020a, Alon et al., 2020] on improved approximation algorithms compare to a bound on the optimal solution that differs from (2.1) and therefore do not fit into our framework.)

First, we address step (ii). Due to space, this proof can be found in 2.9.2. Note that Theorem 8 extends to the fairness constraint defined by [Bera et al., 2019, Bercea et al., 2019]'s provided a fairlet decomposition in this setting.

Theorem 8. *Given an algorithm that obtains a β-approximation to (2.1) where $\beta \leq 1$, and a fairlet decomposition with maximum fairlet size m_f, there is a $\beta\left(1 - \frac{2m_f}{n}\right)$-approximation for fair hierarchical clustering under the revenue objective.*

Prior work showed that average-linkage is a $1/3$-approximation to (2.1) in the vertex-unweighted case; this proof can be easily modified to show that it is still $1/3$-approximation even with vertex weights. This accounts for step (iii) in our process.

Combined with the fairlet decomposition methods for the two-color case [Chierichetti et al., 2017], which has $m_f = b + r$ for b blue vertices and r red vertices, and for multi-color case (Supplementary Material), which has $m_f \leq 2t - 1$, to address step (i), we have the following.

Corollary 9. *There is polynomial time algorithm that constructs a fair tree that is a $\frac{1}{3}\left(1 - \frac{2m_f}{n}\right)$-approximation for revenue objective, where m_f is the maximum size of fairlets.*

2.5 Optimizing value with fairness

In this section we consider the value objective. As in the revenue objective, we prove that we can reduce fair hierarchical clustering to the problem of finding a good fairlet decomposition for the proposed fairlet objective (2.3), and then use any approximation algorithm for weighted hierarchical clustering with the decomposition as the input. Again, our result applies to [Bera et al., 2019, Bercea et al., 2019]'s fairness constraint if we are given an appropriate fairness decomposition.

Theorem 10. *Given an algorithm that gives a β-approximation to (2.2) where $\beta \leq 1$, and a fairlet decomposition $\mathcal{Y}$ such that $\phi(\mathcal{Y}) \leq \epsilon \cdot d(V)$, there is a $\beta(1-\epsilon)$ approximation for (2.2).*

We complement this result with an algorithm that finds a good fairlet decomposition in polynomial time under the bounded representation fairness constraint with cap α.

Let $R_1, \ldots, R_c$ be the c colors and let $\mathcal{Y} = \{Y_1, Y_2 \ldots\}$ be the fairlet decomposition. Let n_i be the number of points colored R_i in V. Let $r_{i,k}$ denote the number of points colored R_i in the kth fairlet.

Theorem 11. *There exists a local search algorithm that finds a fairlet decomposition $\mathcal{Y}$ with $\phi(\mathcal{Y}) \leq (1+\epsilon)\max_{i,k} \frac{r_{i,k}}{n_i} d(V)$ in time $\tilde{O}(n^3/\epsilon)$.*

We can now use the fact that both average-linkage and the $\frac{\epsilon}{n}$-locally-densest cut algorithm give a $\frac{2}{3}$- and $(\frac{2}{3}-\epsilon)$-approximation respectively for vertex-weighted hierarchical clustering under the value objective. Finally, recall that fairlets are intended to be minimal, and their size depends only on the parameter α, and not on the size of the original input. Therefore, as long as the number of points of each color increases as input size, n, grows, the ratio $r_{i,k}/n_i$ goes to 0. These

results, combined with Theorem 10 and Theorem 11, yield Corollary 12.

Corollary 12. *Given bounded size fairlets, the fairlet decomposition computed by local search combined with average-linkage constructs a fair hierarchical clustering that is a* $\frac{2}{3}(1 - o(1))$*-approximation for the value objective. For the* $\frac{\epsilon}{n}$*-locally-densest cut algorithm in [Cohen-Addad et al., 2018], we get a polynomial time algorithm for fair hierarchical clustering that is a* $(\frac{2}{3} - \epsilon)(1 - o(1))$*-approximation under the value objective for any* $\epsilon > 0$*.*

Given at most a small fraction of every color is in any cluster, Corollary 12 states that we can extend the state-of-the-art results for value to the α-capped, multi-colored constraint. Note that the preconditions will always be satisfied and the extension will hold in the two-color fairness setting or in the multi-colored equal representation fairness setting.

Fairlet decompositions via local search In this section, we give a local search algorithm to construct a fairlet decomposition, which proves Theorem 11. This is inspired by the ϵ-densest cut algorithm of [Cohen-Addad et al., 2018]. To start, recall that for a pair of sets A and B we denote by $d(A, B)$ the sum of interpoint distances, $d(A, B) = \sum_{u \in A, v \in B} d(u, v)$. A fairlet decomposition is a partition of the input $\{Y_1, Y_2, \ldots\}$ such that each color composes at most an α fraction of each Y_i.

We start by finding an arbitrary α-capped fairlet decomposition. For two colors with $\alpha = r/(b + r)$, we use the fairlet decomposition introduced by Chierichetti et al. [Chierichetti et al., 2017]. For multiple colors with $\alpha = 1/t$, we defer to Lemma 24 in Appendix 2.9.3.2. Our algorithm will then recursively subdivide the cluster of all data to construct a hierarchy by finding cuts. To search for a cut, we will use a *swap* method.

Definition 13 (Local optimality)**.** *Consider any fairlet decomposition* $\mathcal{Y} = \{Y_1, Y_2, \ldots\}$ *and* $\epsilon > 0$*. Define a* swap *of* $u \in Y_i$ *and* $v \in Y_j$ *for* $j \neq i$ *as updating* Y_i *to be* $(Y_i \setminus \{u\}) \cup \{v\}$ *and* Y_j *to be* $(Y_j \setminus \{v\}) \cup \{u\}$*. We say* $\mathcal{Y}$ *is* ϵ-locally-optimal *if any swap with* u, v *of the same color reduces the objective value by less than a* $(1+\epsilon)$ *factor.*

The algorithm constructs a (ϵ/n)-locally optimal algorithm for fairlet decomposition, which runs in $\tilde{O}(n^3/\epsilon)$ time. Consider any given instance (V, d). Let $d_{\max}$ denote the maximum distance, m_f denote the maximum fairlet size, and $\Delta = d_{\max} \cdot \frac{m_f}{n}$. The algorithm begins with an arbitrary decomposition. Then it swaps pairs of monochromatic points until it terminates with a locally optimal solution. By construction we have the following.

Claim 14. *Algorithm 1 finds a valid fairlet decomposition.*

We prove two things: Algorithm 1 optimizes the objective (2.3), and has a small running time. The following lemma gives an upper bound on $\mathcal{Y}$'s performance for (2.3) found by Algorithm 1.

Algorithm 1 Algorithm for (ϵ/n)-locally-optimal fairlet decomposition.

Input A set V with distance function $d \geq 0$, parameter α, small constant $\epsilon \in [0, 1]$
Output An α-capped fairlet decomposition $\mathcal{Y}$.

1: Find $d_{\max}$, $\Delta \leftarrow \frac{m_f}{n} d_{\max}$.
2: Arbitrarily find an α-capped fairlet decomposition $\{Y_1, Y_2, \ldots\}$ such that each partition has at most an α fraction of any color ▷ See [Chierichetti et al., 2017] or Appendix 2.9.3.2
3: **while** $\exists u \in Y_i, v \in Y_j, i \neq j$ of the same color, such that for the decomposition $\mathcal{Y}'$ after swapping u, v, $\frac{\sum_{Y_k \in \mathcal{Y}} d(Y_k)}{\sum_{Y_k \in \mathcal{Y}'} d(Y_k)} \geq (1 + \epsilon/n)$ **and** $\sum_{Y_k \in \mathcal{Y}} d(Y_k) > \Delta$ **do**
4: Swap u and v by setting $Y_i \leftarrow (Y_i \setminus \{u\}) \cup \{v\}$ and $Y_j \leftarrow (Y_j \setminus \{v\}) \cup \{u\}$.
5: **end while**

Lemma 15. *The fairlet decomposition* $\mathcal{Y}$ *computed by Algorithm 1 has an objective value for (2.3) of at most* $(1+\epsilon) \max_{i,k} \frac{r_{i,k}}{n_i} d(V)$.

Finally we bound the running time. The algorithm has much better performance in practice than its worst-case analysis would indicate. We will show this later in Section 2.7.

Lemma 16. *The running time for Algorithm 1 is* $\tilde{O}(n^3/\epsilon)$.

Together, Lemma 15, Lemma 16, and Claim 14 prove Theorem 11. This establishes that there is a local search algorithm that can construct a good fairlet decomposition.

2.6 Optimizing cost with fairness

This section considers the cost objective of [Dasgupta, 2016]. Even without our fairness constraint, the difficulty of approximating cost is clear in its approximation hardness and the fact that all known solutions require an LP or SDP solver. We obtain the result in Theorem 17; extending this result to other fairness constraints, improving its bound, or even making the algorithm practical, are open questions.

Theorem 17. *Consider the two-color case. Given a* β*-approximation for cost and a* γ_t*-approximation for minimum weighted bisection* [2] *on input of size* t*, then for parameters* t *and* ℓ *such that* $n \geq t\ell$ *and* $n > \ell + 108t^2/\ell^2$*, there is a fair* $O\left(\frac{n}{t} + t\ell + \frac{n\ell\gamma_t}{t} + \frac{nt\gamma_t}{\ell^2}\right)$ β*-approximation for* $\mathrm{cost}(T^*_{\mathrm{unfair}})$.

With proper parameterization, we achieve an $O\left(n^{5/6}\log^{5/4} n\right)$-approximation. We defer our algorithm description, pseudocode, and proofs to the Supplementary Material. While our algorithm is not simple, it is an important (and non-obvious) step to show the existence of an approximation, which we hope will spur future work in this area.

[2] The minimum weighted bisection problem is to find a partition of nodes into two equal-sized subsets so that the sum of the weights of the edges crossing the partition is minimized.

Table 2.1: Dataset description. Here (b, r) denotes the balance of the dataset.

Name	Sample size	# features	Protected feature	Color (blue, red)	(b, r)
CENSUSGENDER	30162	6	gender	(female, male)	$(1, 3)$
CENSUSRACE	30162	6	race	(non-white, white)	$(1, 7)$
BANKMARRIAGE	45211	7	marital status	(not married, married)	$(1, 2)$
BANKAGE	45211	7	age	$(< 40, \geq 40)$	$(2, 3)$

Table 2.2: Impact of Algorithm 1 on $\text{ratio}_{\text{value}}$ in percentage (mean $\pm$ std. dev).

Samples	400	800	1600	3200	6400	12800
CENSUSGENDER, initial	88.17 ± 0.76	88.39 ± 0.21	88.27 ± 0.40	88.12 ± 0.26	88.00 ± 0.10	88.04 ± 0.13
final	99.01 ± 0.60	99.09 ± 0.58	99.55 ± 0.26	99.64 ± 0.13	99.20 ± 0.38	99.44 ± 0.23
CENSUSRACE, initial	84.49 ± 0.66	85.01 ± 0.31	85.00 ± 0.42	84.88 ± 0.43	84.84 ± 0.16	84.89 ± 0.20
final	99.50 ± 0.20	99.89 ± 0.32	100.0 ± 0.21	99.98 ± 0.21	99.98 ± 0.11	99.93 ± 0.31
BANKMARRIAGE, initial	92.47 ± 0.54	92.58 ± 0.30	92.42 ± 0.30	92.53 ± 0.14	92.59 ± 0.14	92.75 ± 0.04
final	99.18 ± 0.22	99.28 ± 0.33	99.59 ± 0.14	99.51 ± 0.17	99.46 ± 0.10	99.50 ± 0.05
BANKAGE, initial	93.70 ± 0.56	93.35 ± 0.41	92.95 ± 0.25	93.28 ± 0.13	93.36 ± 0.12	93.33 ± 0.12
final	99.40 ± 0.28	99.40 ± 0.51	99.61 ± 0.13	99.64 ± 0.07	99.65 ± 0.08	99.59 ± 0.06

2.7 Experiments

This section validates our algorithms from Sections 2.4 and 2.5 empirically. We adopt the disparate impact fairness constraint [Chierichetti et al., 2017]; thus each point is either blue or red. In particular, we would like to:

- Show that running the standard average-linkage algorithm results in highly unfair solutions.
- Demonstrate that demanding fairness in hierarchical clustering incurs only a small loss in the hierarchical clustering objective.
- Show that our algorithms, including fairlet decomposition, are practical on real data.

In 2.9.7 we consider multiple colors and the same trends as the two color case occur.

Datasets. We use two datasets from the UCI data repository.[3] In each dataset, we use features with numerical values and leave out samples with empty entries. For value, we use the Euclidean distance as the dissimilarity measure. For revenue, we set the similarity to be $s(i, j) = \frac{1}{1+d(i,j)}$

[3] `archive.ics.uci.edu/ml/index.php`, Census: `archive.ics.uci.edu/ml/datasets/census+income`, Bank: `archive.ics.uci.edu/ml/datasets/Bank+Marketing`

where $d(i, j)$ is the Euclidean distance. We pick two different protected features for both datasets, resulting in four datasets in total (See Table 2.1 for details).

- *Census* dataset: We choose *gender* and *race* to be the protected feature and call the resulting datasets CENSUSGENDERand CENSUSRACE.
- *Bank* dataset: We choose *marital status* and *age* to be the protected features and call the resulting datasets BANKMARRIAGEand BANKAGE.

In this section, unless otherwise specified, we report results only for the value objective. Results for the revenue objective are qualitatively similar and are omitted here. We do not evaluate our algorithm for the cost objective since it is currently only of theoretical interest.

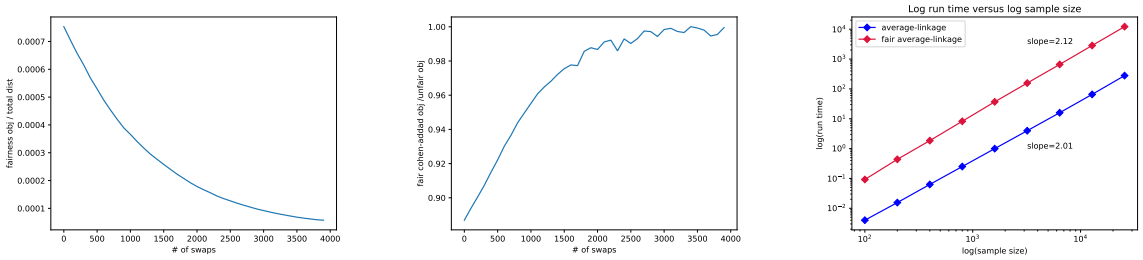
We sub-sample points of two colors from the original data set proportionally, while approximately retaining the original color balance. The sample sizes used are $100 \times 2^i, i = 0, \ldots, 8$. On each, we do 5 experiments and report the average results. We set ϵ in Algorithm 1 to 0.1 in all of the experiments.

Implementation. The code is available in the Supplementary Material. In the experiments, we use Algorithm 1 for the fairlet decomposition phase, where the fairlet decomposition is initialized by randomly assigning red and blue points to each fairlet. We apply the average-linkage algorithm to create a tree on the fairlets. We further use average-linkage to create subtrees inside of each fairlet.

The algorithm selects a *random* pair of blue or red points in different fairlets to swap, and checks if the swap sufficiently improves the objective. We do not run the algorithm until all the pairs are checked, rather the algorithm stops if it has made a $2n$ failed attempts to swap a random pair. As we observe empirically, this does not have material effect on the quality of the overall

Table 2.3: Impact of Algorithm 1 on $\text{ratio}_{\text{fairlets}}$.

Samples	100	200	400	800	1600	3200	6400	12800
CENSUSGENDER, initial	2.5e-2	1.2e-2	6.2e-3	3.0e-3	1.5e-3	7.5e-4	3.8e-4	1.9e-4
final	4.9e-3	1.4e-3	6.9e-4	2.5e-4	8.5e-5	3.6e-5	1.8e-5	8.0e-6
CENSUSRACE, initial	6.6e-2	3.4e-2	1.7e-2	8.4e-3	4.2e-3	2.1e-3	1.1e-3	5.3e-4
final	2.5e-2	1.2e-2	6.2e-3	3.0e-3	1.5e-3	7.5e-4	3.8e-4	1.9e-5
BANKMARRIAGE, initial	1.7e-2	8.2e-3	4.0e-3	2.0e-3	1.0e-3	5.0e-4	2.5e-4	1.3e-4
final	5.9e-3	2.1e-3	9.3e-4	4.1e-4	1.3e-4	7.1e-5	3.3e-5	1.4e-5
BANKAGE, initial	1.3e-2	7.4e-3	3.5e-3	1.9e-3	9.3e-4	4.7e-4	2.3e-4	1.2e-4
final	5.0e-3	2.2e-3	7.0e-4	3.7e-4	1.3e-4	5.7e-5	3.0e-5	1.4e-5



(i) (ii) (iii)

Figure 2.1: (i) $\text{ratio}_{\text{fairlets}}$, every 100 swaps. (ii) $\text{ratio}_{\text{value}}$, every 100 swaps. (iii) CENSUSGENDER: running time vs sample size on a log-log scale.

solution.

Metrics. We present results for value here, the results for revenue are qualitatively similar. In our experiments, we track the following quantities. Let G be the given input instance and let T be the output of our fair hierarchical clustering algorithm. We consider the following ratio $\text{ratio}_{\text{value}} = \frac{\text{value}_G(T)}{\text{value}_G(T')}$, where T' is the tree obtained by the standard average-linkage algorithm. We consider the fairlet objective function where $\mathcal{Y}$ is a fairlet decomposition. Let $\text{ratio}_{\text{fairlets}} = \frac{\phi(\mathcal{Y})}{d(V)}$.

Results. Average-linkage algorithm always constructs unfair trees. For each of the datasets, the algorithm results in monochromatic clusters at some level, strengthening the case for fair algorithms.

In Table 2.2, we show for each dataset the $\text{ratio}_{\text{value}}$ both at the time of initialization (Initial) and after using the local search algorithm (Final). We see the change in the ratio as the local

Table 2.4: Clustering on fairlets found by local search vs. upper bound, at size 1600 (mean $\pm$ std. dev).

Dataset	CENSUSGENDER	CENSUSRACE	BANKMARRIAGE	BANKAGE
Revenue vs. upper bound	81.89 ± 0.40	81.75 ± 0.83	61.53 ± 0.37	61.66 ± 0.66
Value vs. upper bound	84.31 ± 0.15	84.52 ± 0.22	89.17 ± 0.29	88.81 ± 0.18

search algorithm performs swaps. Fairness leads to almost no degradation in the objective value as the swaps increase. Table 2.3 shows the $\text{ratio}_{\text{fairlets}}$ between the initial initialization and the final output fairlets. As we see, Algorithm 1 significantly improves the fairness of the initial random fairlet decomposition. The more the locally-optimal algorithm improves the objective value of (2.3), the better the tree's performance based on the fairlets. Figures 2.1(i) and 2.1(ii) show $\text{ratio}_{\text{value}}$ and $\text{ratio}_{\text{fairlets}}$ for every 100 swaps in the execution of Algorithm 1 on a subsample of size 3200 from Census data set. The plots show that as the fairlet objective value decreases, the value objective of the resulting fair tree increases. Such correlation are found on subsamples of all sizes.

Now we compare the objective value of the algorithm with the upper bound on the optimum. We report the results for both the revenue and value objectives, using fairlets obtained by local search, in Table 2.4. On all datasets, we obtain ratios significantly better than the theoretical worst case guarantee. In Figure 2.1(iii), we show the average running time on Census data for both the original average-linkage and the fair average-linkage algorithms. As the sample size grows, the running time scales almost as well as current implementations of average-linkage algorithm. Thus with a modest increase in time, we can obtain a fair hierarchical clustering under the value objective.

2.8 Conclusions

In this paper we extended the notion of fairness to the classical problem of hierarchical clustering under three different objectives (revenue, value, and cost). Our results show that revenue and value are easy to optimize with fairness; while optimizing cost appears to be more challenging.

Our work raises several questions and research directions. Can the approximations be improved? Can we find better upper and lower bounds for fair cost? Are there other important fairness criteria?

Broader Impact

Our work builds upon a long line of work of fairness in machine learning. See the excellent books by Kearns and Roth [Kearns and Roth, 2020], and Barocas et al. [Barocas et al., 2019] for a rich introduction to the field.

Our aim in this work is algorithmic in nature, finding near-optimal hierarchical clustering algorithms that attain certain fairness guarantees. Since these methods are common unsupervised learning primitives, it is important to develop tools for practitioners to use. At the same time we remark that just because an algorithm is proven to be "fair" under some definition, does not mean it can be applied blindly.

As is now well known, [Kleinberg et al., 2017b], different fairness notions can be incompatible with each other. Moreover, fairness in machine learning is necessarily problem specific, and depends on the goals and the values of the person invoking the algorithm. While these facts are well established in the research community, they are far from common knowledge outside of it. Thus work on algorithmic notions of fairness runs the risk of someone treating the results as a silver bullet, and eschewing the deeper analysis that is necessary in any real world application.

Funding Sources

B. Moseley and Y. Wang were supported in part by a Google Research Award, an Infor Research Award, a Carnegie Bosch Junior Faculty Chair and NSF grants CCF-1824303, CCF-1845146, CCF-1733873 and CMMI-1938909. B. Moseley additionally is a part time employee of Relational-AI. M. Knittel was supported in part by NSF BIGDATA grant IIS-1546108, and NSF

SPX grant CCF-1822738 and some of the work was conducted while she was visiting Google.

2.9 Appendix

2.9.1 Approximation algorithms for weighted hierarchical clustering

In this section we first prove that running constant-approximation algorithms on fairlets gives good solutions for value objective, and then give constant approximation algorithms for both revenue and value in weighted hierarchical clustering problem, as is mentioned in Corollary 9 and 12. That is, a weighted version of average-linkage, for both weighted revenue and value objective, and weighted (ϵ/n)-locally densest cut algorithm, which works for weighted value objective. Both proofs are easily adapted from previous proofs in [Cohen-Addad et al., 2018] and [Moseley and Wang, 2017].

2.9.1.1 Running constant-approximation algorithms on fairlets

In this section, we prove Theorem 10, which says if we run any β-approximation algorithm for the upper bound on weighted value on the fairlet decomposition, we get a fair tree with minimal loss in approximation ratio. For the remainder of this section, fix any hierarchical clustering algorithm A that is guaranteed on any *weighted* input (V, d, m) to construct a hierarchical clustering with objective value at least $\beta m(V)d(V)$ for the value objective on a weighted input. Recall that we extended the value objective to a weighted variant in the Preliminaries Section and $m(V) = \sum_{u \in V} m_u$. Our aim is to show that we can combine A with the fairlet decomposition $\mathcal{Y}$ introduced in the prior section to get a fair hierarchical clustering that is a $\beta(1-\epsilon)$-approximation for the value objective, if $\phi(\mathcal{Y}) \leq \epsilon d(V)$.

In the following definition, we transform the point set to a new set of points that are

weighted. We will analyze A on this new set of points. We then show how we can relate this to the objective value of the optimal tree on the original set of points.

Definition 18. *Let $\mathcal{Y} = \{Y_1, Y_2, \ldots\}$ be the fairlet decomposition for V that is produced by the local search algorithm. Define $V(\mathcal{Y})$ as follows:*

- *Each set Y_i has a corresponding point a_i in $V(\mathcal{Y})$.*
- *The weight m_i of a_i is set to be $|Y_i|$.*
- *For each partitions Y_i, Y_j, where $i \neq j$ and $Y_i, Y_j \in \mathcal{Y}$, $d(a_i, a_j) = d(Y_i, Y_j)$.*

We begin by observing the objective value that A receives on the instance $V(\mathcal{Y})$ is large compared to the weights in the original instance.

Theorem 19. *On the instance $V(\mathcal{Y})$ the algorithm A has a total weighted objective of $\beta(1-\epsilon) \cdot nd(V)$.*

Proof. Notice that $m(V(\mathcal{Y})) = |V| = n$. Consider the total sum of all the distances in $V(\mathcal{Y})$. This is $\sum_{a_i, a_j \in V(\mathcal{Y})} d(a_i, a_j) = \sum_{Y_i, Y_j \in \mathcal{Y}} d(Y_i, Y_j) = d(V) - \phi(\mathcal{Y})$. The upper bound on the optimal solution is $(\sum_{Y_i \in \mathcal{Y}} m_i)(d(V) - \phi(\mathcal{Y}) = n(d(V) - \phi(\mathcal{Y}))$. Since $\phi(\mathcal{Y}) \leq \epsilon d(V)$, this upper bound is at least $(1-\epsilon)nd(V)$. Theorem 10 follows from the fact that the algorithm A archives a weighted revenue at least a β factor of the total weighted distances. □

2.9.1.2 Weighted hierarchical clustering: Constant-factor approximation

For weighted hierarchical clustering with positive integral weights, we define the weighted average-linkage algorithm for input (V, d, m) and (V, s, m). Define the *average distance* to be

$Avg(A,B) = \frac{d(A,B)}{m(A)m(B)}$ for dissimilarity-based input, and $Avg(A,B) = \frac{s(A,B)}{m(A)m(B)}$ for similarity-based input. In each iteration, weighted average-linkage seeks to merge the clusters which minimizes this value, if dissimilarity-based, and maximizes this value, if similarity-based.

Lemma 20. *Weighted average-linkage is a* $\frac{2}{3}(resp., \frac{1}{3})$ *approximation for the upper bound on weighted value (resp., revenue) objective with positive, integral weights.*

Proof. We prove it for weighted value first. This is directly implied by the fact that average-linkage is $\frac{2}{3}$ approximation for unweighted value objective, as is proved in [Cohen-Addad et al., 2018]. We have already seen in the last subsection that a unweighted input V can be converted into weighted input $V(\mathcal{Y})$. Vice versa, we can construct a weighted input (V, d, m) into unweighted input with same upper bound for value objective.

In weighted hierarchical clustering we treat each point p with integral weights as $m(p)$ duplicates of points with distance 0 among themselves, let's call this set $S(p)$. For two weighted points $(p, m(p))$ and $(q, m(q))$, if $i \in S(p), j \in S(q)$, let $d(i,j) = \frac{d(p,q)}{m(p)m(q)}$. This *unweighted* instance, composed of many duplicates, has the same upper bound as the weighted instance. Notice that running average-linkage on the unweighted instance will always choose to put all the duplicates $S(p)$ together first for each p, and then do hierarchical clustering on top of the duplicates. Thus running average-linkage on the unweighted input gives a valid hierarchical clustering tree for weighted input. Since unweighted value upper bound equals weighted value upper bound, the approximation ratio is the same.

Now we prove it for weighted revenue. In [Moseley and Wang, 2017], average-linkage being $\frac{1}{3}$ approximation for unweighted revenue is proved by the following. Given any clustering

$\mathcal{C}$, if average-linkage chooses to merge A and B in $\mathcal{C}$, we define a local revenue for this merge:

$$\text{merge-rev}(A, B) = \sum_{C \in \mathcal{C} \setminus \{A,B\}} |C||A||B|Avg(A, B).$$

And correspondingly, a local cost:

$$\text{merge-cost}(A, B) = \sum_{C \in \mathcal{C} \setminus \{A,B\}} (|B||A||C|Avg(A, C) + |A||B||C|Avg(B, C)).$$

Summing up the local revenue and cost over all merges gives the upper bound. [Moseley and Wang, 2017] used the property of average-linkage to prove that at every merge, merge-cost$(A, B) \leq$ 2merge-rev(A, B), which guarantees the total revenue, which is the summation of merge-rev(A, B) over all merges, is at least $\frac{1}{3}$ of the upper bound. For the weighted case, we define

$$\text{merge-rev}(A, B) = \sum_{C \in \mathcal{C} \setminus \{A,B\}} m(C)m(A)m(B)Avg(A, B).$$

And

$$\begin{aligned}\text{merge-cost}(A, B) = \sum_{C \in \mathcal{C} \setminus \{A,B\}} & (m(B)m(A)m(C)Avg(A, C) \\ & + m(A)m(B)m(C)Avg(B, C)).\end{aligned}$$

And the rest of the proof works in the same way as in [Moseley and Wang, 2017], proving weighted average-linkage to be $\frac{1}{3}$ for weighted revenue. □

Next we define the weighted (ϵ/n)-locally-densest cut algorithm. The original algorithm,

introduced in [Cohen-Addad et al., 2018], defines a cut to be $\frac{d(A,B)}{|A||B|}$. It starts with the original set as one cluster, at every step, it seeks the partition of the current set that locally maximizes this value, and thus constructing a tree from top to bottom. For the weighted input (V, d, m), we define the cut to be $\frac{d(A,B)}{m(A)m(B)}$, and let $n = m(V)$. For more description of the algorithm, see Algorithm 4 in Section 6.2 in [Cohen-Addad et al., 2018].

Lemma 21. *Weighted (ϵ/n)-locally-densest cut algorithm is a $\frac{2}{3} - \epsilon$ approximation for weighted value objective.*

Proof. Just as in the average-linkage proof, we convert each weighted point p into a set S of $m(p)$ duplicates of p. Notice that the converted unweighted hierarchical clustering input has the same upper bound as the weighted hierarchical clustering input, and the ϵ/n-locally-densest cut algorithm moves all the duplicate sets S around in the unweighted input, instead of single points as in the original algorithm in [Cohen-Addad et al., 2018].

Focus on a split of cluster $A \cup B$ into (A, B). Let S be a duplicate set. $\forall S \subseteq A$, where S is a set of duplicates, we must have

$$(1 + \frac{\epsilon}{n})\frac{d(A, B)}{|A||B|} \geq \frac{d(A \setminus S, B \cup S)}{(|A| - |S|)(|B| + |S|)}.$$

Pick up a point $q \in S$,

$$
\begin{aligned}
&(1+\frac{\epsilon}{n})d(A,B)|S|(|A|-1)(|B|+1) \\
&= (1+\frac{\epsilon}{n})d(A,B)(|A||B|+|A|-|B|-1)|S| \\
&= (1+\frac{\epsilon}{n})d(A,B)(|A||B|+|A||S|-|B||S|-|S|)+(1+\frac{\epsilon}{n})d(A,B)(|A||B|)(|S|-1) \\
&\geq (1+\frac{\epsilon}{n})d(A,B)(|A|-|S|)(|B|+|S|)+d(A,B)|A||B|(|S|-1) \\
&\geq |A||B|d(A\setminus S, B\cup S)+d(A,B)|A||B|(|S|-1) \\
&= |A||B|(d(A,B)+|S|d(q,A)-|S|d(q,B))+|A||B|(|S|-1)d(A,B) \\
&= |A||B||S|(d(A,B)+d(q,A)-d(q,B)).
\end{aligned}
$$

Rearrange the terms and we get the following inequality holds for any point $q \in A$:

$$
\left(1+\frac{\epsilon}{n}\right)\frac{d(A,B)}{|A||B|} \geq \frac{d(A,B)+d(q,A)-d(q,B)}{(|A|-1)(|B|+1)}.
$$

The rest of the proof goes exactly the same as the proof in [Cohen-Addad et al., 2018, Theorem 6.5]. □

2.9.2 Proof of Theorem 8

Proof. Let $\mathcal{A}$ be the β-approximation algorithm to (2.1). For a given instance $G = (V, s)$, let $\mathcal{Y} = \{Y_1, Y_2, \ldots\}$ be a fairlet decomposition of V; let $m_f = \max_{Y\in\mathcal{Y}} |Y|$. Recall that $n = |V|$.

We use $\mathcal{Y}$ to create a weighted instance $G_{\mathcal{Y}} = (\mathcal{Y}, s_{\mathcal{Y}}, m_{\mathcal{Y}})$. For $Y, Y' \in \mathcal{Y}$, we define $s(Y, Y') = \sum_{i\in Y, j\in Y'} s(i, j)$ and we define $m_{\mathcal{Y}}(Y) = |Y|$.

We run $\mathcal{A}$ on $G_{\mathcal{Y}}$ and let $T_{\mathcal{Y}}$ be the hierarchical clustering obtained by $\mathcal{A}$. To extend this

to a tree T on V, we simply place all the points in each fairlet as leaves under the corresponding vertex in $T_{\mathcal{Y}}$.

We argue that $\mathrm{rev}_G(T) \geq \beta\left(1 - \frac{2m_f}{n}\right)(n-2)s(V)$.

Since $\mathcal{A}$ obtains a β-approximation to hierarchical clustering on $G_{\mathcal{Y}}$, we have $\mathrm{rev}_{G_{\mathcal{Y}}}\left(T_{\mathcal{Y}}\right) \geq \beta \cdot \sum_{Y,Y' \in \mathcal{Y}} s(Y,Y')(n - m(Y) - m(Y'))$.

Notice the fact that, for any pair of points u, v in the same fairlet $Y \in \mathcal{Y}$, the revenue they get in the tree T is $(n - m(Y))s(u,v)$. Then using $rev_G(T) = \sum_{Y \in \mathcal{Y}}(n - m(Y))s(Y) + \mathrm{rev}(T_{\mathcal{Y}})$,

$$\begin{aligned}\mathrm{rev}_G(T) &\geq \sum_{Y \in \mathcal{Y}} \beta(n - m(Y))s(Y) + \beta \sum_{Y,Y' \in \mathcal{Y}} s(Y,Y')(n - m(Y) - m(Y')) \\ &\geq \beta(n - 2m_f)\left(\sum_{Y \in \mathcal{Y}} s(Y) + \sum_{Y,Y' \in \mathcal{Y}} s(Y,Y')\right) \geq \beta\left(1 - \frac{2m_f}{n}\right)(n-2)s(V).\end{aligned}$$

Thus the resulting tree T is a $\beta\left(1 - \frac{2m_f}{n}\right)$-approximation of the upper bound. □

2.9.3 Proofs for (ϵ/n)locally-optimal local search algorithm

In this section, we prove that Algorithm 1 gives a good fairlet decomposition for the fairlet decomposition objective 2.3, and that it has polynomial run time.

2.9.3.1 Proof for a simplified version of Lemma 15

In Subsection 2.9.3.2, we will prove Lemma 15. For now, we will consider a simpler version of Lemma 15 in the context of [Chierichetti et al., 2017]'s disparate impact problem, where we have red and blue points and strive to preserve their ratios in all clusters. Chierichetti et al. [Chierichetti et al., 2017] provided a valid fairlet decomposition in this context, where each

fairlet has at most b blue points and r red points. Before going deeper into the analysis, we state the following useful proposition.

Proposition 22. *Let $r_t = |red(V)|$ be the total number of red points and $b_t = |blue(V)|$ the number of blue points. We have that, $\max\{\frac{r}{r_t}, \frac{b}{b_t}\} \leq \frac{2(b+r)}{n}$.*

Proof. Recall that $balance(V) = \frac{b_t}{r_t} \geq \frac{b}{r}$, and wlog $b_t \leq r_t$. Since the fractions are positive and $\frac{b_t}{r_t} \geq \frac{b}{r}$ we know that $\frac{b_t}{b_t+r_t} \geq \frac{b}{b+r}$. Since $b_t + r_t = n$ we conclude that $b_t \geq \frac{b}{b+r}n$. Similarly, we conclude that $\frac{r_t}{b_t+r_t} \leq \frac{r}{b+r}$. Therefore $r_t \leq \frac{r}{b+r}n$.

Thus, $\frac{r}{r_t} \geq \frac{b+r}{n} \geq \frac{b}{b_t}$. However, since $b_t \leq r_t$ and $b_t + r_t = n$, $r_t \geq \frac{1}{2}n$, $\frac{r}{r_t} \leq \frac{2r}{n} \leq \frac{2(b+r)}{n}$. □

Using this, we can define and prove the following lemma, which is a simplified version of Lemma 15.

Lemma 23. *The fairlet decomposition $\mathcal{Y}$ computed by Algorithm 1 has an objective value for (2.3) of at most $(1+\epsilon)\frac{2(b+r)}{n}d(V)$.*

Proof. Let $Y : V \mapsto \mathcal{Y}$ denote a mapping from a point in V to the fairlet it belongs to. Let $d_R(X) = \sum_{u \in red(X)} d(u, X)$, and $d_B(X) = \sum_{v \in blue(X)} d(v, X)$. Naturally, $d_R(X) + d_B(X) = 2d(X)$ for any set X. For a fairlet $Y_i \in \mathcal{Y}$, let r_i and b_i denote the number of red and blue points in Y_i.

We first bound the total number of intra-fairlet pairs. Let $x_i = |Y_i|$, we know that $0 \leq x_i \leq b+r$ and $\sum_i x_i = n$. The number of intra-fairlet pairs is at most $\sum_i x_i^2 \leq \sum_i (b+r)x_i = (b+r)n$.

The **While** loop can end in two cases: 1) if $\mathcal{Y}$ is (ϵ/n)-locally-optimal; 2) if $\sum_{Y_k \in \mathcal{Y}} d(Y_k) \leq \Delta$. Case 2 immediately implies the lemma, thus we focus on case 1. By definition of the algorithm, we know that for any pair $u \in Y(u)$ and $v \in Y(v)$ where u, v have the same color and

$Y(u) \neq Y(v)$ the swap does not increase objective value by a large amount. (The same trivially holds if the pair are in the same cluster.)

$$\sum_{Y_k} d(Y_k) \leq (1+\frac{\epsilon}{n})(\sum_{Y_k} d(Y_k) - d(u,Y(u)) - d(v,Y(v)) + d(u,Y(v)) + d(v,Y(u)) - 2d(u,v))$$
$$\leq (1+\frac{\epsilon}{n})(\sum_{Y_k} d(Y_k) - d(u,Y(u)) - d(v,Y(v)) + d(u,Y(v)) + d(v,Y(u))).$$

After moving terms and some simplification, we get the following inequality:

$$\begin{aligned} & d(u,Y(u)) + d(v,Y(v)) \\ \leq\ & d(u,Y(v)) + d(v,Y(u)) + \frac{\epsilon/n}{1+\epsilon/n} \sum_{Y_k \in \mathcal{Y}} d(Y_k) \\ \leq\ & d(u,Y(v)) + d(v,Y(u)) + \frac{\epsilon}{n} \sum_{Y_k \in \mathcal{Y}} d(Y_k). \end{aligned} \tag{2.4}$$

Then we sum up (2.4), $d(u,Y(u))+d(v,Y(v)) \leq d(u,Y(v))+d(v,Y(u))+\frac{\epsilon}{n}\sum_{Y_k\in\mathcal{Y}} d(Y_k)$, over every pair of points in $red(V)$ (even if they are in the same partition).

$$r_t \sum_{Y_i} d_R(Y_i) \leq \left(\sum_{Y_i} r_i d_R(Y_i)\right) + \left(\sum_{u \in red(V)} \sum_{Y_i \neq Y(u)} r_i d(u,Y_i)\right) + r_t^2 \frac{\epsilon}{n} \sum_{Y_i} d(Y_i).$$

Divide both sides by r_t and use the fact that $r_i \leq r$ for all Y_i:

$$\sum_{Y_i} d_R(Y_i) \leq \left(\sum_{Y_i} \frac{r}{r_t} d_R(Y_i)\right) + \left(\sum_{u \in red(V)} \sum_{Y_i \neq Y(u)} \frac{r}{r_t} d(u,Y_i)\right) + \frac{r_t \epsilon}{n} \sum_{Y_i} d(Y_i). \tag{2.5}$$

For pairs of points in $blue(V)$ we sum (2.4) to similarly obtain:

$$\sum_{Y_i} d_B(Y_i) \leq \left(\sum_{Y_i} \frac{b}{b_t} d_B(Y_i) \right) + \left(\sum_{v \in blue(V)} \sum_{Y_i \neq Y(v)} \frac{b}{b_t} d(v, Y_i) \right) + \frac{b_t \epsilon}{n} \sum_{Y_i} d(Y_i). \tag{2.6}$$

Now we sum up (2.5) and (2.6). The LHS becomes:

$$\sum_{Y_i} (d_R(Y_i) + d_B(Y_i)) = \sum_{Y_i} \sum_{u \in Y_i} d(u, Y_i) = 2 \sum_{Y_i} d(Y_i.)$$

For the RHS, the last term in (2.5) and (2.6) is $\frac{\epsilon(b_t + r_t)}{n} \sum_{Y_i} d(Y_i) = \epsilon \sum_{Y_i} d(Y_i)$.

The other terms give:

$$\begin{aligned}
&\frac{r}{r_t} \sum_{Y_i} d_R(Y_i) + \frac{r}{r_t} \sum_{u \in red(V)} \sum_{Y_i \neq Y(u)} d(u, Y_i) + \frac{b}{b_t} \sum_{Y_i} d_B(Y_i) + \frac{b}{b_t} \sum_{v \in blue(V)} \sum_{Y_i \neq Y(v)} d(v, Y_i) \\
&\leq \max\{\frac{r}{r_t}, \frac{b}{b_t}\} \left\{ \sum_{Y_i} (d_R(Y_i) + d_B(Y_i)) + \sum_{u \in V} \sum_{Y_i \neq Y(u)} d(u, Y_i) \right\} \\
&= \max\{\frac{r}{r_t}, \frac{b}{b_t}\} \left\{ \sum_{Y_i} \sum_{u \in Y_i} d(u, Y_i) + \sum_{Y_i} \sum_{Y_j \neq Y_i} d(Y_i, Y_j) \right\} \\
&= 2 \max\{\frac{r}{r_t}, \frac{b}{b_t}\} d(V) \\
&\leq \frac{4(b+r)}{n} d(V).
\end{aligned}$$

The last inequality follows from Proposition 22. All together, this proves that

$$2 \sum_{Y_k} d(Y_k) \leq \frac{4(b+r)}{n} d(V) + \epsilon \sum_{Y_k} d(Y_k).$$

Then, $\frac{\sum_{Y_k} d(Y_k)}{d(V)} \leq \frac{2(b+r)}{n} \cdot \frac{1}{1-\epsilon/2} \leq (1+\epsilon)\frac{2(b+r)}{n}$. The final step follows from the fact that

$(1+\epsilon)(1-\epsilon/2) = 1 + \frac{\epsilon}{2}(1-\epsilon) \geq 1$. This proves the lemma. □

2.9.3.2 Proof for the generalized Lemma 15

Next, we prove Lemma 15 for the more generalized definition of fairness, which is α-capped fairness.

Proof of [Lemma 15] The proof follows the same logic as in the two-color case: we first use the (ϵ/n)-local optimality of the solution, and sum up the inequality over all pairs of points with the same color.

Let $Y : V \mapsto \mathcal{Y}$ denote a mapping from a point in V to the fairlet it belongs to. Let $R_i(X)$ be the set of R_i colored points in a set X. Let $d_{R_i}(X) = \sum_{u \in R_i(X)} d(u, X)$. Naturally, $\sum_i d_{R_i}(x) = 2d(X)$ for any set X since the weight for every pair of points is repeated twice.

The **While** loop can end in two cases: 1) if $\mathcal{Y}$ is (ϵ/n)-locally-optimal; 2) if $\sum_{Y_k \in \mathcal{Y}} d(Y_k) \leq \Delta$. Case 2 immediately implies the lemma, thus we focus on case 1.

By definition of the algorithm, we know that for any pair $u \in Y(u)$ and $v \in Y(v)$ where u, v have the same color and $Y(u) \neq Y(v)$ the swap does not increase objective value by a large amount. (The same trivially holds if the pair are in the same cluster.) We get the following inequality as in the two color case:

$$d(u, Y(u)) + d(v, Y(v)) \leq d(u, Y(v)) + d(v, Y(u)) + \frac{\epsilon}{n} \sum_{Y_k \in \mathcal{Y}} d(Y_k). \tag{2.7}$$

For any color R_i, we sum it over every pair of points in $R_i(V)$ (even if they are in the same

partition).

$$n_i \sum_{Y_k} d_{R_i}(Y_k) \leq \left(\sum_{Y_k} r_{ik} d_{R_i}(Y_k) \right) + \left(\sum_{u \in R_i(V)} \sum_{Y_k \neq Y(u)} r_{ik} d(u, Y_k) \right) + n_i^2 \frac{\epsilon}{n} \sum_{Y_k} d(Y_k).$$

Divide both sides by n_i and we get:

$$\sum_{Y_k} d_{R_i}(Y_k) \leq \left(\sum_{Y_k} \frac{r_{ik}}{n_i} d_{R_i}(Y_k) \right) + \left(\sum_{u \in R_i(V)} \sum_{Y_k \neq Y(u)} \frac{r_{ik}}{n_i} d(u, Y_k) \right) + \frac{n_i \epsilon}{n} \sum_{Y_k} d(Y_k). \quad (2.8)$$

Now we sum up this inequality over all colors R_i. The LHS becomes:

$$\sum_{Y_k} \sum_{i} d_{R_i}(Y_k) = \sum_{Y_k} \sum_{u \in Y_k} d(u, Y_k) = 2 \sum_{Y_k} d(Y_k).$$

For the RHS, the last term sums up to $\frac{\epsilon(\sum_i n_i)}{n} \sum_{Y_k} d(Y_k) = \epsilon \sum_{Y_k} d(Y_k)$. Using the fact that $\frac{r_{ik}}{n_i} \leq \max_{i,k} \frac{r_{ik}}{n_i}$, the other terms sum up to :

$$\begin{aligned}
&\sum_{i} \sum_{Y_k} \frac{r_{ik}}{n_i} d_{R_i}(Y_k) + \sum_{i} \sum_{u \in R_i(V)} \sum_{Y_k \neq Y(u)} \frac{r_{ik}}{n_i} d(u, Y_k) \\
&\leq \max_{i,k} \frac{r_{ik}}{n_i} \left\{ \sum_{Y_k} \sum_{i} d_{R_i}(Y_i) + \sum_{u \in V} \sum_{Y_k \neq Y(u)} d(u, Y_k) \right\} \\
&= \max_{i,k} \frac{r_{ik}}{n_i} \left\{ \sum_{Y_k} \sum_{u \in Y_k} d(u, Y_k) + \sum_{Y_k} \sum_{Y_j \neq Y_k} d(Y_j, Y_k) \right\} \\
&= 2 \max_{i,k} \frac{r_{ik}}{n_i} \cdot d(V).
\end{aligned}$$

Therefore, putting LHS and RHS together, we get

$$2\sum_{Y_k} d(Y_k) \leq 2\max_{i,k} \frac{r_{ik}}{n_i} d(V) + \epsilon \sum_{Y_k} d(Y_k).$$

Then, $\frac{\sum_{Y_k} d(Y_k)}{d(V)} \leq \max_{i,k} \frac{r_{ik}}{n_i} \cdot \frac{1}{1-\epsilon/2} \leq (1+\epsilon) \cdot \max_{i,k} \frac{r_{ik}}{n_i}$. The final step follows from the fact that $(1+\epsilon)(1-\epsilon/2) = 1 + \frac{\epsilon}{2}(1-\epsilon) \geq 1$.

□

In the two-color case, the ratio $\max_{i,k} \frac{r_{ik}}{n_i}$ becomes $\max\{\frac{r}{r_t}, \frac{b}{b_t}\}$, which can be further bounded by $\frac{2(b+r)}{n}$ (see Proposition 22). If there exists a caplet decomposition such that $\max_{i,k} \frac{r_{ik}}{n_i} = o(1)$, Lemma 15 implies we can build a fair hierarchical clustering tree with $o(1)$ loss in approximation ratio for value objective.

Assuming for all color class R_i, $n_i \to +\infty$ as $n \to +\infty$, here we give a possible caplet decomposition for $\alpha = \frac{1}{t}(t <= c)$ with size $O(t)$ for positive integer t, thus guaranteeing $\max_{i,k} \frac{r_{ik}}{n_i} = o(1)$ for any i.

Lemma 24. *For any set P of size p that satisfies fairness constraint with $\alpha = 1/t$, there exists a partition of P into sets $(P_1, P_2, \ldots)$ where each P_i satisfies the fairness constraint and $t \leq |P_i| < 2t$.*

Proof. Let $p = m \times t + r$ with $0 \leq r < t$, then the fairness constraints ensures that there are at most m elements of each color. Consider partitioning obtained through the following process: consider an ordering of elements where points of the same color are in consecutive places, assign points to sets $P_1, P_2, \ldots, P_m$ in a round robin fashion. So each set P_i gets at least t elements and at most $t + r < 2t$ elements assigned to it. Since there are at most m elements of each color,

each set gets at most one point of any color and hence all sets satisfy the fairness constraint as $1 \leq \frac{1}{t} \cdot |P_i|$. □

2.9.3.3 Proof for the running time of (ϵ/n)locally-optimal fairlet decomposition algorithm

Proof of [Lemma 16] Notice that finding the maximum pairwise distance takes $O(n^2)$ time. Thus, we focus on analyzing the time spent on the **While** loop.

Let t be the total number of swaps. We argue that $t = \tilde{O}(n/\epsilon)$. If $t = 0$ the conclusion trivially holds. Otherwise, consider the decomposition $\mathcal{Y}_{t-1}$ before the last swap. Since the **While** loop does not terminate here, $\sum_{Y_k \in \mathcal{Y}_{t-1}} d(Y_k) \geq \Delta = \frac{b+r}{n} d_{max}$. However, at the beginning, we have $\sum_{Y_k \in \mathcal{Y}} d(Y_k) \leq (b+r)n \cdot d_{max} = n^2 \Delta \leq n^2 \sum_{Y_k \in \mathcal{Y}_{t-1}} d(Y_k)$. Therefore, it takes at most $\log_{1+\epsilon/n}(n^2) = \tilde{O}(n/\epsilon)$ iterations to finish the **While** loop.

It remains to discuss the running time of each iteration. We argue that there is a way to finish each iteration in $O(n^2)$ time. Before the **While** loop, keep a record of $d(u, Y_i)$ for each point u and each fairlet Y_i. This takes $O(n^2)$ time. If we know $d(u, Y_i)$ and the objective value from the last iteration, in the current iteration, it takes $O(1)$ time to calculate the new objective value after each swap (u, v), and there are at most n^2 such calculations, before the algorithm either finds a pair to swap, or determines that no such pair is left. After the swap, the update for all the $d(u, Y_i)$ data takes $O(n)$ time. In total, every iteration takes $O(n^2)$ time.

Therefore, Algorithm 1 takes $\tilde{O}(n^3/\epsilon)$ time. □

2.9.4 Hardness of optimal fairlet decomposition

Before proving Theorem 7, we state that the PARTITION INTO TRIANGLES (PIT) problem is known to belong to the NP-complete class [Garey and Johnson, 2002], defined as follows. In the definition, we call a clique k-clique if it has k nodes. A triangle is a 3-clique.

Definition 25. PARTITION INTO TRIANGLES (PIT). *Given graph $G = (V, E)$, where $V = 3n$, determine if V can be partitioned into 3-element sets $S_1, S_2, \ldots, S_n$, such that each S_i forms a triangle in G.*

The NP-hardness of PIT problem gives us a more general statement.

Definition 26. PARTITION INTO k-CLIQUES (PIKC). *For a fixed number k treated as constant, given graph $G = (V, E)$, where $V = kn$, determine if V can be partitioned into k-element sets $S_1, S_2, \ldots, S_n$, such that each S_i forms a k-clique in G.*

Lemma 27. *For a fixed constant $k \geq 3$, the PIKC problem is NP-hard.*

Proof. We reduce the PIKC problem from the PIT problem. For any graph $G = (V, E)$ given to the PIT problem where $|V| = 3n$, construct another graph $G' = (V', E')$. Let $V' = V \cup C_1 \cup C_2 \cup \cdots \cup C_n$, where all the C_i's are $(k-3)$-cliques, and there is no edge between any two cliques C_i and C_j where $i \neq j$. For any C_i, let all points in C_i to be connected to all nodes in V.

Now let G' be the input to PIKC problem. We prove that G can be partitioned into triangles if and only if G' can be partitioned into k-cliques. If V has a triangle partition $V = \{S_1, \ldots, S_n\}$, then $V' = \{S_1 \cup C_1, \ldots, S_n \cup C_n\}$ is a k-clique partition. On the other hand, if V' has a k-clique partition $V' = \{S'_1, \ldots, S'_n\}$ then $C_1, \ldots, C_n$ must each belong to different k-cliques

since they are not connected to each other. Without loss of generality we assume $C_i \subseteq S_i$, then $V = \{S'_1 \setminus C_1, \ldots, S'_n \setminus C_n\}$ is a triangle partition. □

We are ready to prove the theorem.

Proof of [Theorem 7] We prove Theorem 7 by proving that for given $z \geq 4$, if there exists a c-approximation polynomial algorithm $\mathcal{A}$ for (2.3), it can be used to solve the PIKC problem where $k = z - 1$ for any instance as well. This holds for any finite c.

Given any graph $G = (V, E)$ that is input to the PIKC problem, where $|V| = kn = (z-1)n$, let a set V' with distances be constructed in the following way:

1. $V' = V \cup \{C_1, \ldots, C_n\}$, where each C_i is a singleton.

2. Color the points in V red, and color all the C_i's blue.

3. For a $e = (u, v)$, let $d(u, v) = 0$, if it satisfies one of the three conditions: 1) $e \in E$. 2) $u, v \in C_i$ for some C_i. 3) one of u, v is in V, while the other belong to some C_i.

4. All other edges have distance 1.

Obviously the blue points make up a $1/z$ fraction of the input so each fairlet should have exactly 1 blue point and $z - 1$ red points.

We claim that G has a k-clique partition if and only if algorithm $\mathcal{A}$ gives a solution of 0 for (2.3). The same argument as in the proof of Lemma 27 will show that G has a k-clique partition if and only if the optimal solution to (2.3) is 0. This is equal to algorithm $\mathcal{A}$ giving a solution of 0 since otherwise the approximate is not bounded. □

2.9.5 Optimizing cost with fairness

In this section, we present our fair hierarchical clustering algorithm that approximates Dasgupta's cost function and satisfies Theorem 17. Most of the proofs can be found in Section 2.9.5.1. We consider the problem of equal representation, where vertices are red or blue and $\alpha = 1/2$. From now on, whenever we use the word "fair", we are referring to this fairness constraint. Our algorithm also uses parameters t and ℓ such that $n \geq t\ell$ and $t > \ell + 108t^2/\ell^2$ for $n = |V|$, and leverages a β-approximation for cost and γ_t-approximation for minimum weighted bisection. We will assume these are fixed and use them throughout the section.

We will ultimately show that we can find a fair solution that is a sublinear approximation for the unfair optimum T^*_{unfair}, which is a lower bound of the fair optimum. Our main result is Theorem 17, which is stated in the body of the paper.

The current best approximations described in Theorem 17 are $\gamma_t = O(\log^{3/2} n)$ by [Feige and Krauthgamer, 2000] and $\beta = \sqrt{\log n}$ by both [Dasgupta, 2016] and [Charikar and Chatziafratis, 2017]. If we set $t = \sqrt{n}(\log^{3/4} n)$ and $\ell = n^{1/3}\sqrt{\log n}$, then we get Corollary 28.

Corollary 28. *Consider the equal representation problem with two colors. There is an* $O\left(n^{5/6}\log^{5/4} n\right)$*-approximate fair clustering under the cost objective.*

The algorithm will be centered around a single clustering, which we call $\mathcal{C}$, that is extracted from an unfair hierarchy. We will then adapt this to become a similar, fair clustering $\mathcal{C}'$. To formalize what $\mathcal{C}'$ must satisfy to be sufficiently "similar" to $\mathcal{C}$, we introduce the notion of a $\mathcal{C}$-good clustering. Note that this is not an intuitive set of properties, it is simply what $\mathcal{C}'$ must satisfy in order

Definition 29 (Good clustering)**.** *Fix a clustering $\mathcal{C}$ whose cluster sizes are at most t. A fair clustering $\mathcal{C}'$ is* $\mathcal{C}$-good *if it satisfies the following two properties:*

1. *For any cluster $C \in \mathcal{C}$, there is a cluster $C' \in \mathcal{C}'$ such that all but (at most) an $O(\ell\gamma_t/t + t\gamma_t/\ell^2)$-fraction of the weight of edges in C is also in C'.*

2. *Any $C' \in \mathcal{C}'$ is not too much bigger, so $|C'| \leq 6t\ell$.*

The hierarchy will consist of a $\mathcal{C}$-good (for a specifically chosen $\mathcal{C}$) clustering $\mathcal{C}'$ as its only nontrivial layer.

Lemma 30. *Let T be a β-approximation for cost and $\mathcal{C}$ be a maximal clustering in T under the condition that all cluster sizes are at most t. Then, a fair two-tiered hierarchy T' whose first level consists of a $\mathcal{C}$-good clustering achieves an $O\left(\frac{n}{t} + t\ell + \frac{n\ell\gamma_t}{t} + \frac{nt\gamma_t}{\ell^2}\right)$ β-approximation for cost.*

Proof. Since T is a β-approximation, we know that:

$$\operatorname{cost}(T) \leq \beta \operatorname{cost}(T^*_{\text{unfair}})$$

We will then utilize a scheme to account for the cost contributed by each edge relative to their cost in T in the hopes of extending it to T^*_{unfair}. There are three different types of edges:

1. An edge e that is merged into a cluster of size t or greater in T, thus contributing $t \cdot s(e)$ to the cost. At worst, this edge is merged in the top cluster in T' to contribute $n \cdot s(e)$. Thus, the factor increase in the cost contributed by e is $O(n/t)$. Then since the total contribution of all such edges in T is at most $\operatorname{cost}(T)$, the total contribution of all such edges in T' is at most $O(n/t) \cdot \operatorname{cost}(T)$.

2. An edge e that started in some cluster $C \in \mathcal{C}$ that does not remain in the corresponding cluster C'. We are given that the total weight removed from any such C is an $O(\ell\gamma_t/t + t\gamma_t/\ell^2)$-fraction of the weight contained in C. If we sum across the weight in all clusters in $\mathcal{C}$, that is at most $\text{cost}(T)$. So the total amount of weight moved is at most $O(\ell\gamma_t/t + t\gamma_t/\ell^2)\cdot\text{cost}(T)$. These edges contributed at least $2s(e)$ in T as the smallest possible cluster size is two. In T', these may have been merged at the top of the cluster, for a maximum cost contribution of $n \cdot s(e)$. Therefore, the total cost across all such edges is increased by at most a factor of $n/2$, which gives a total cost of at most $O(n\ell\gamma_t/t + nt\gamma_t/\ell^2) \cdot \text{cost}(T)$.

3. An edge e that starts in some cluster $C \in \mathcal{C}$ and remains in the corresponding $C' \in \mathcal{C}'$. Similarly, this must have contributed at least $2s(e)$ in T, but now we know that this edge is merged within C' in T', and that the size of C' is $|C'| \leq 6t\ell$. Thus its contribution increases at most by a factor of $3t\ell$. By the same reasoning from the first edge type we discussed, all these edges total contribute at most a factor of $O(t\ell) \cdot \text{cost}(T)$.

We can then put a conservative bound by putting this all together.

$$\text{cost}(T') \leq O\left(\frac{n}{t} + t\ell + \frac{n\ell\gamma_t}{t} + \frac{nt\gamma_t}{\ell^2}\right)\text{cost}(T).$$

Finally, we know T is a β-approximation for T^*_{unfair}.

$$\text{cost}(T') \leq O\left(\frac{n}{t} + t\ell + \frac{n\ell\gamma_t}{t} + \frac{nt\gamma_t}{\ell^2}\right) \cdot \beta \cdot \text{cost}(T^*_{\text{unfair}}).$$

□

With this proof, the only thing left to do is find a $\mathcal{C}$-good clustering $\mathcal{C}'$ (Definition 29).

Specifically, using the clustering $\mathcal{C}$ mentioned in Lemma 30, we would like to find a $\mathcal{C}$-good clustering $\mathcal{C}'$ using the following.

Lemma 31. *There is an algorithm that, given a clustering $\mathcal{C}$ with maximum cluster size t, creates a $\mathcal{C}$-good clustering.*

The proof is deferred to the Section 2.9.5.1. With these two Lemmas, we can prove Theorem 17.

Proof. Consider our graph G. We first obtain a β-approximation for unfair cost, which yields a hierarchy tree T. Let $\mathcal{C}$ be the maximal clustering in T under the constraint that the cluster sizes must not exceed t. We then apply the algorithm from Lemma 31 to get a $\mathcal{C}$-good clustering $\mathcal{C}'$. Construct T' such that it has one layer that is $\mathcal{C}'$. Then we can apply the results from Lemma 30 to get the desired approximation. □

From here, we will only provide a high-level description of the algorithm for Lemma 31. For precise details and proofs, see Section 2.9.5.1. To start, we need to propose some terminology.

Definition 32 (Red-blue matching)**.** *A **red-blue matching** on a graph G is a matching M such that $M(u) = v$ implies u and v are different colors.*

Red-blue matchings are interesting because they help us ensure fairness. For instance, suppose M is a red-blue matching that is also perfect (i.e., touches all nodes). If the lowest level of a hierarchy consists of a clustering such that v and $M(v)$ are in the same cluster for all v, then that level of the hierarchy is fair since there is a bijection between red and blue vertices within each cluster. When these clusters are merged up in the hierarchy, fairness is preserved.

Our algorithm will modify an unfair clustering to be fair by combining clusters and moving a small number of vertices. To do this, we will use the following notion.

Definition 33 (Red-blue clustering graph)**.** *Given a graph G and a clustering $\mathcal{C} = \{C_1, \ldots, C_k\}$, we can construct a **red-blue clustering graph** $H_M = (V_M, E_M)$ that is associated with some red-blue matching M. Then H_M is a graph where $V_M = \mathcal{C}$ and $(C_i, C_j) \in E_M$ if and only if there is a $v_i \in C_i$ and $M(v_i) = v_j \in C_j$.*

Basically, we create a graph of clusters, and there is an edge between two clusters if and only if there is at least one vertex in one cluster that is matched to some vertex in the other cluster. We now show that the red-blue clustering graph can be used to construct a fair clustering based on an unfair clustering.

Proposition 34. *Let H_M be a red-blue clustering graph on a clustering $\mathcal{C}$ with a perfect red-blue matching M. Let $\mathcal{C}'$ be constructed by merging all the clusters in each component of H_M. Then $\mathcal{C}'$ is fair.*

Proof. Consider some $C \in \mathcal{C}'$. By construction, this must correspond to a connected component in H_M. By definition of H_M, for any vertex $v \in C$, $M(v) \in C$. That means M, restricted to C, defines a bijection between the red and blue nodes in C. Therefore, C has an equal number of red and blue vertices and hence is fair. □

We will start by extracting a clustering $\mathcal{C}$ from an unfair hierarchy T that approximates cost. Then, we will construct a red-blue clustering graph H_M with a perfect red-blue matching M. Then we can use the components of H_M to define our first version of the clustering $\mathcal{C}'$. However, this requires a non-trivial way of moving vertices between clusters in $\mathcal{C}$.

We now give an overview of our algorithm in Steps (A)–(G). For a full description, see our pseudocode in Section 2.9.8.

(A) **Get an unfair approximation** T. We start by running a β-approximation for cost in the unfair setting. This gives us a tree T such that $\text{cost}(T) \leq \beta \cdot \text{cost}(T^*_{\text{unfair}})$.

(B) **Extract a t-maximal clustering**. Given T, we find the maximal clustering $\mathcal{C}$ such that (i) every cluster in the clustering is of size at most t, and (ii) any cluster above these clusters in T is of size more than t.

(C) **Combine clusters to be size t to $3t$.** We will now slowly change $\mathcal{C}$ into $\mathcal{C}'$ during a number of steps. In the first step, we simply define $\mathcal{C}_0$ by merging small clusters $|C| \leq t$ until the merged size is between t and $3t$. Thus clusters in $\mathcal{C}$ are contained within clusters in $\mathcal{C}_0$, and all clusters are between size t and $3t$.

(D) **Find cluster excesses.** Next, we strive to make our clustering more fair. We do this by trying to find an underlying matching between red and blue vertices that agrees with $\mathcal{C}_0$ (matches are in the same cluster). If the matching were perfect, then the clusters in $\mathcal{C}_0$ would have equal red and blue representation. However, this is not guaranteed initially. We start by conceptually matching as many red and blue vertices within clusters as we can. Note we do not actually create this matching; we just want to reserve the space for this matching to ensure fairness, but really some of these vertices may be moved later on. Then the remaining unmatched vertices in each cluster is either entirely red or entirely blue. We call this amount the *excess* and the color the *excess color*. We label each cluster with both of these.

(E) **Construct red-blue clustering graph**. Next, we would like to construct $H_M = (V_M, E_M)$, our red-blue clustering graph on $\mathcal{C}_0$. Let $V_M = \mathcal{C}_0$. In addition, for the within-cluster matchings mentioned in Step (D), let those matches be contained in M. With this start, we will

do a matching process to simultaneously construct E_M and the rest of M. Note the unmatched vertices are specifically the excess vertices in each cluster. We will match these with an iterative process given our parameter ℓ:

1. Select a vertex $C_i \in V_M$ with excess at least ℓ to start a new connected component in H_M. Without loss of generality, say its excess color is red.

2. Find a vertex $C_j \in V_M$ whose excess color is blue and whose excess is at least ℓ. Add (C_i, C_j) to E_M.

3. Say without loss of generality that the excess of C_i is less than that of C_j. Then match all the excess in C_i to vertices in the excess of C_j. Now C_j has a smaller excess.

4. If C_j has an excess less than ℓ or C_j is the ℓth cluster in this component, end this component. Start over at (1) with a new cluster.

5. Otherwise, use C_j as our reference and continue constructing this component at (2).

6. Complete when there are no more clusters with over ℓ excess that are not in a component (or all remaining such clusters have the same excess color).

We would like to construct $\mathcal{C}'$ by merging all clusters in each component. This would be fair if M were a perfect matching, however this is not true yet. In the next step, we handle this.

(F) **Fix unmatched vertices.** We now want to match excess vertices that are unmatched. We do this by bringing vertices from other clusters into the clusters that have unmatched excess, starting with all small unmatched excess. Note that some clusters were never used in Step (E) because they had small excess to start. This means they had many internal red-blue matches. Remove t^2/ℓ^2 of these and put them into clusters in need. For other vertices, we will later describe

a process where t/ℓ of the clusters can contribute $108t^2/\ell^2$ vertices to account for unmatched excess. Thus clusters lose at most $108t^2/\ell^2$ vertices, and we account for all unmatched vertices. Call the new clustering $\mathcal{C}_1$. Now M is perfect and H_M is unchanged.

(G) **Define $\mathcal{C}'$.** Finally, we create the clustering $\mathcal{C}'$ by merging the clusters in each component of H_M. Note that Proposition 34 assures C' is fair. In addition, we will show that cluster sizes in $\mathcal{C}_1$ are at most $6t$, so $\mathcal{C}'$ has the desired upper bound of $6t\ell$ on cluster size. Finally, we removed at most $\ell + t^2/\ell^2$ vertices from each cluster. This is the desired $\mathcal{C}$-good clustering.

Further details and the proofs that the above sequence of steps achieve the desired approximation can be found in the next section. While the approximation factor obtained is not as strong as the ones for revenue or value objectives with fairness, we believe cost is a much harder objective with fairness constraints.

2.9.5.1 Proof of Theorem 17

This algorithm contains a number of components. We will discuss the claims made by the description step by step. In Step (A), we simply utilize any β-approximation for the unfair approximation. Step (B) is also quite simple. At this point, all that is left is to show how to find $\mathcal{C}'$, ie, prove Lemma 31 (introduced in Section 2.6). This occurs in the steps following Step (B). In Step (C), we apply our first changes to the starting clustering from T. We now prove that the cluster sizes can be enforced to be between t and $3t$.

Lemma 35. *Given a clustering $\mathcal{C}$, we can construct a clustering $\mathcal{C}_0$, where each $C \in \mathcal{C}_0$ is a union of clusters in $\mathcal{C}$ and $t \leq |C| < 3t$.*

Proof. We iterate over all clusters in $\mathcal{C}$ whose size are less than t and continually merge them

until we create a cluster of size $\geq t$. Note that since the last two clusters we merged were of size $< t$, this cluster is of size $t \leq |C| < 2t$. We then stop this cluster and continue merging the rest of the clusters. At the end, if we are left with a single cluster of size $< t$, we simply merge this with any other cluster, which will then be of size $t \leq |C| < 3t$. □

Step (D) describes a rather simple process. All we have to do in each cluster is count the amount of each color in each cluster, find which is more, and also compute the difference. No claims are made here.

Step (E) defines a more careful process. We describe this process and its results here.

Lemma 36. *There is an algorithm that, given a clustering $\mathcal{C}_0$ with $t \leq |C| \leq 3t$ for $C \in \mathcal{C}_0$, can construct a red-blue clustering graph $H_M = (V_M, E_M)$ on $\mathcal{C}_0$ with underlying matching M such that:*

1. *H_M is a forest, and its max component size is ℓ.*
2. *For every $(C_i, C_j) \in E_M$, there are at least ℓ matches between C_i and C_j in M. In other words, $|M(C_i) \cap C_j| \geq \ell$.*
3. *For most $C_i \in V_M$, at most ℓ vertices in C_i are unmatched in M. The only exceptions to this rule are (1) exactly one cluster in every ℓ-sized component in H_M, and (2) at most $n/2$ additional clusters.*

Proof. We use precisely the process from Step 5. Let $V_M = \mathcal{C}_0$. H_M will look like a bipartite graph with some entirely isolated nodes. We then try to construct components of H_M one-by-one such that (1) the max component size is ℓ, and (2) edges represent at least ℓ matches in M.

Let us show it satisfies the three conditions of the lemma. For condition 1, note that we will always halt component construction once it reaches size ℓ. Thus no component can exceed size ℓ. In addition, for every edge added to the graph, at least one of its endpoints now has small excess and will not be considered later in the program. Thus no cycles can be created, so it is a forest.

For condition 2, consider the construction of any edge $(C_i, C_j) \in E_M$. At this point, we only consider C_i and C_j to be clusters with different-color excess of at least ℓ each. In the next part of the algorithm, we match as much excess as we can between the two clusters. Therefore, there must be at least ℓ underlying matches.

Finally, condition 3 will be achieved by the completion condition. By the completion condition, there are no isolated vertices (besides possibly those leftover of the same excess color) that have over ℓ excess. Whenever we add a cluster to a component, either that cluster matches all of its excess, or the cluster it becomes adjacent to matches all of its excess. Therefore at any time, any component has at most one cluster with any excess at all. If the component is smaller than ℓ (and is not the final component), then that can only happen when in the final addition, both clusters end up with less than ℓ excess. Therefore, no cluster in this component can have less than ℓ excess. For an ℓ-sized component, by the rule mentioned before, only one cluster can remain with ℓ excess. When the algorithm completes, we are left with a number of large-excess clusters with the same excess color, say red without loss of generality. Assume for contradiction there are more than $n/2$ such clusters, and so there is at least $n\ell/2$. Since we started with half red and half blue vertices, the remaining excess in the rest of the clusters must match up with the large red excess. Thus the remaining at most $n/2$ clusters must have at least $n\ell/2$ blue excess, but this is only achievable if they have large excess left. This is a contradiction. Thus we satisfy condition

3. □

This concludes Step (E). In Step (F), we will transform the underlying clustering $\mathcal{C}_0$ such that we can achieve a perfect matching M. This will require removing a small number of vertices from some clusters in $\mathcal{C}_0$ and putting them in clusters that have unmatched vertices. This process will at most double cluster size.

Lemma 37. *There is an algorithm that, given a clustering $\mathcal{C}_0$ with $t \leq |C| \leq 3t$ for $C \in \mathcal{C}_0$, finds a clustering $\mathcal{C}_1$ and an underlying matching M' such that:*

1. *There is a bijection between $\mathcal{C}_0$ and $\mathcal{C}_1$.*
2. *For any cluster $C_0 \in \mathcal{C}_0$ and its corresponding $C_1 \in \mathcal{C}_1$, $|C_0| - |C_1| \leq \ell + 108t^2/\ell^2$. This means that at most ℓ vertices are removed from C_0 in the construction of C_1.*
3. *For all $C_1 \in \mathcal{C}_1$, $t - \ell - 108t^2/\ell^2 \leq |C_1| \leq 6t$.*
4. *M' is a perfect red-blue matching.*
5. *H_M is a red-blue clustering graph of $\mathcal{C}_1$ with matching M', perhaps with additional edges.*

Proof. Use Lemma 36 to find the red-blue clustering graph H_M and its corresponding graph M. Then we know that only one cluster in every ℓ-sized component plus one other cluster can have a larger than ℓ excess. Since cluster sizes are at least t, $|V_M| \geq n/t$. This means that at most $n/(t\ell) + 1 = (n + t\ell)/(t\ell) \leq 2n/(t\ell)$ clusters need more than ℓ vertices. Since the excess is upper bounded by cluster size which is upper bounded by $3t$, this is at most $6n/\ell$ vertices in large excess that need matches.

We will start by removing all small excess vertices from clusters. This removes at most ℓ from any cluster. These vertices will then be placed in clusters with large excess of the right color. If we run out of large excess of the right color that needs matches, since the total amount of red and blue vertices is balanced, that means we can instead transfer the unmatched small excess red vertices to clusters with a small amount of unmatched blue vertices. In either case, this accounts for all the small unmatched excess. Now all we need to account for is at most $6n/\ell$ unmatched vertices in large excess clusters. At this point, note that the large excess should be balanced between red and blue. From now on, we will remove matches from within and between clusters to contribute to this excess. Since this always contributes the same amount of red and blue vertices by breaking matches, we do not have to worry about the balance of colors. We will describe how to distribute these contributions across a large number of clusters.

Consider vertices that correspond to clusters that (ignoring the matching M) started out with at most ℓ excess. So the non-excess portion, which is at least size $t - \ell$, is entirely matched with itself. We will simply remove t^2/ℓ^2 of these matches to contribute.

Otherwise, we will consider vertices that started out with large excess. We must devise a clever way to break matches without breaking too many incident upon a single cluster. For every tree in H_M (since H_M is a forest by Lemma 36), start at the root, and do a breadth-first search over all internal vertices. At any vertex we visit, break ℓ matches between it and its child (recall by by Lemma 36 that each edge in H_M represents at least ℓ inter-cluster matches). Thus, each break contributes 2ℓ vertices. We do this for every internal vertex. Since an edge represents at least ℓ matches and the max cluster size is at most $3t$, any vertex can have at most $3t/\ell$ children. Thus the fraction of vertices in H_M that correspond to a contribution of 2ℓ vertices is at least $\ell/(3t)$.

Clearly, the worst case is when all vertices in H_M have large excess, as this means that fewer clusters are ensured to be able to contribute. By Lemma 36, at least $n/2$ of these are a part of completed connected components (ie, of size ℓ or with each cluster having small remaining excess). So consider this case. Since $|V_M| \geq n/(3t)$, then this process yields $n\ell^2/(18t^2)$ vertices. To achieve $6n/\ell$ vertices, we must then run $108t^2/\ell^3$ iterations. If an edge no longer represents ℓ matches because of an earlier iteration, consider it a non-edge for the rest of the process. The only thing left to consider is if a cluster C becomes isolated in H_M during the process. We know C began with at least t vertices, and at most ℓ were removed by removing small excess. So as long as $t > \ell + 108t^2/\ell^2$, we can remove the rest of the $108t^2/\ell^2$ vertices from the non-excess in C (the rest must be non-excess) in the same way as vertices that were isolated in H_M to start. Thus, we can account for the entire set of unmatched vertices without removing more than $108t^2/\ell^2$ vertices from any given cluster.

Now we consider the conditions. Condition 1 is obviously satisfied because we are just modifying clusters in $\mathcal{C}_0$, not removing them. The second condition is true because of our careful accounting scheme where we only remove $\ell + 108t^2/\ell^2$ vertices per cluster. The same is true for the lower bound in condition 3. When we add them to new clusters, since we only add a vertex to match an unmatched vertex, we at most double cluster size. So the max cluster size is $6t$.

For the fourth condition, note that we explicitly executed this process until all unmatched vertices became matched, and any endpoint in a match we broke was used to create a new match. Thus the new matching, which we call M', is perfect. It is still red-blue. Finally, note we did not create any matches between clusters. Therefore, no match in M' can violate H_M. Thus condition 5 is met. □

Finally, we construct our final clustering in Step (G). However, to satisfy the qualities of Lemma 30, we must first argue about the weight loss from each cluster.

Lemma 38. *Consider any clustering $\mathcal{C}$ with cluster sizes between t and $6t$. Say each cluster has a specified r number of red vertices to remove and b number of blue vertices to remove such that $r + b \leq x$ for some x, and r (resp. b) is nonzero only if the number of red (resp. blue) vertices in the cluster is $O(n)$. Then we can remove the desired number of each color while removing at most an $O((x/t)\gamma_t)$ of the weight originally contained within the cluster.*

Proof. Consider some cluster C with parameters r and b. We will focus first on removing red vertices. Let C_r be the red vertex set in C. We create a graph K corresponding to this cluster as follows. Let b_0 be a vertex representing all blue vertices from C, b_0' be the "complement" vertex to b_0, and R be a set of vertices r_i corresponding to all red vertices in C. We also add a set of $2r-|C_r|+2X$ dummy vertices (where X is just some large value that makes it so $2r-|C_r|+X > 0$). $2r-|C_r|+X$ of the dummy vertices will be connected to b_0 with infinite edge weight (denote these δ_i), the other X will be connected to b_0' with infinite edge weight (denote these δ_i'). This will ensure that b_0 and b_0' are in the same partitions as their corresponding dummies. Let s_G and s_K be the similarity function in the original graph and new graph respectively.

$$s_K(b_0, \delta_i) = \infty$$

$$s_K(b_0', \delta_i') = \infty$$

The blue vertex b_0 is also connected to all r_i with the following weight (where C_b is the set

of blue vertices in C):

$$s_K(b_0, r_i) = \sum_{b_j \in C_b} s_G(r_i, b_j) + \frac{1}{2} \sum_{r_j \in R \setminus \{r_j\}} s_G(r_i, r_j)$$

This edge represents the cumulative edge weight between r_i and all blue vertices. The additional summation term, which contains the edge weights between r_i and all other red vertices, is necessary to ensure our bisection cut will also contain the edge weights between two of the removed red vertices.

Next, the edge weights between red vertices must contain the other portion of the corresponding edge weight in the original graph.

$$s_K(r_i, r_j) = \frac{1}{2} s_G(r_i, r_j)$$

Now, we note that there are a total of $2 - |C_r| + 2X + |C_r| = 2r + 2X$ vertices. So a bisection will partition the graph into vertex sets of size $r + X$. Obviously, in any approximation, b_0 must be grouped with all δ_i and b_0' must be grouped with all δ_i'. This means the b_0 partition must contain $|C_r| - r$ of the R vertices, and the b_0' partition must contain the other r. These r vertices in the latter partition are the ones we select to move.

Consider any set S of r red vertices in K. Then it is a valid bisection. We now show that the edge weight in the cut for this bisection is exactly the edge weight lost by removing S from K. We can do this algebraically. We start by breaking down the weight of the cut into the weight between the red vertices in S and b_0, and also the red vertices in S and the red vertices not in S.

$$
\begin{aligned}
s_K(S, V(K) \setminus S) &= \sum_{r_i \in S} s_K(b_0, r_i) + \sum_{r_i \in S, r_j \in R \setminus S} s_K(r_i, r_j) \\
&= \sum_{r_i \in S} \left(\sum_{b_j \in B} s_G(r_i, b_j) + \frac{1}{2} \sum_{r_j \in R \setminus \{r_j\}} s_G(r_i, r_j) \right) \\
&\quad + \sum_{r_i \in S, r_j \in R \setminus S} \frac{1}{2} s_G(r_i, r_j) \\
&= \sum_{r_i \in S} \left(\sum_{b_j \in B} s_G(r_i, b_j) + \frac{1}{2} \sum_{r_j \in R \setminus \{r_j\}} s_G(r_i, r_j) \right. \\
&\qquad \left. + \frac{1}{2} \sum_{r_j \in R \setminus S} s_G(r_i, r_j) \right)
\end{aligned}
$$

Notice that the two last summations have an overlap. They both contribute half the edge weight between r_i and vertices in $R \setminus S$. Thus, these edges contribute their entire edge weight. All remaining vertices in $S \setminus \{r_i\}$ only contribute half their edge weight. We can then distribute the summation.

$$
\begin{aligned}
s_K(S, V(K) \setminus S) &= \sum_{r_i \in S} \left(\sum_{b_j \in B} s_G(r_i, b_j) + \frac{1}{2} \sum_{r_j \in S \setminus \{r_j\}} s_G(r_i, r_j) \right. \\
&\qquad \left. + \sum_{r_j \in R \setminus S} s_G(r_i, r_j) \right) \\
&= \sum_{r_i \in S, b_j \in B} s_G(r_i, b_j) + \frac{1}{2} \sum_{r_i \in S, r_j \in S \setminus \{r_j\}} s_G(r_i, r_j) \\
&\quad + \sum_{r_i \in S, r_j \in R \setminus S} s_G(r_i, r_j)
\end{aligned}
$$

In the middle summation, note that every edge $e = (u, v)$ is counted twice when $r_i = u$ and $r_j = v$, and when $r_i = v$ and $r_j = u$. We can then rewrite this as:

$$s_K(S, V(K) \setminus S) = \sum_{r_i \in S, b_j \in B} s_G(r_i, b_j) + \sum_{r_i, r_j \in S} s_G(r_i, r_j) + \sum_{r_i \in S, r_j \in R \setminus S} s_G(r_i, r_j)$$

When we remove S, we remove the connections between S and blue vertices, the connections within S, and the connections between S and red vertices not in S. This is precisely what this accounts for. Therefore, any bisection on K directly corresponds to removing a vertex set S of r red vertices from C. If we have a γ_t-approximation for minimum weighted bisection, then, this yields a γ_t-approximation for the smallest loss we can achieve from removing r red vertices.

Now we must compare the optimal way to remove r vertices to the total weight in a cluster. Let $\rho = |C_r|$ be the number of red vertices in a cluster. Then the total number of possible cuts to isolate r red vertices is $\binom{\rho}{r}$. Let $\mathcal{S}$ be the set of all possible cuts to isolate r red vertices. Then if we sum over the weight of all possible cuts (where weight here is the weight between the r removed vertices and all vertices, including each other), that will sum over each red-red edge and blue-red edge multiple times. A red-red edge is counted if either of its endpoints is in $S \in \mathcal{S}$, and this happens $2\binom{\rho}{r-1} - \binom{R-1}{r-2} \leq 2\binom{\rho}{r-1}$ of the time. A blue-red edge is counted if its red endpoint is in S, which happens $\binom{\rho}{r-1} \leq 2\binom{\rho}{r-1}$. And of course, since no blue-blue edge is covered, each is covered under $2\binom{\rho}{r-1}$ times. Therefore, if we sum over all these cuts, we get at most $2\binom{\rho}{r-1}$ times the weight of all edges in C.

$$\sum_{S \in \mathcal{S}} s(S) \leq 2\binom{\rho}{r-1} s(C)$$

Let OPT be the minimum possible cut. Now since there are $\binom{\rho}{r}$ cuts, we know the lefthand side here is bounded above by $\binom{\rho}{r}s(OPT)$.

$$\binom{\rho}{r}s(OPT) \leq 2\binom{\rho}{r-1}s(C)$$

We can now simplify.

$$s(OPT) \leq \frac{2r}{\rho}s(C)$$

But note we are given $\rho = O(t)$. So if we have a γ_t approximation for the minimum bisection problem, this means we can find a way to remove r vertices such that the removed weight is at most $O(r/t)\gamma_t$. We can do this again to get a bound on the removal of the blue vertices. This yields a total weight removal of $O(x/t)\gamma_t$. □

Finally, we can prove Lemma 31, which satisfies the conditions of Lemma 30.

Proof. Start by running Lemma 35 on $\mathcal{C}$ to yield $\mathcal{C}_0$. Then we can apply Lemma 37 to yield $\mathcal{C}_1$ with red-blue clustering graph H_M and underlying perfect red-blue matching M'. We create $\mathcal{C}'$ by merging components in H_M into clusters. Since the max component size is ℓ and the max cluster size in $\mathcal{C}_1$ is $6t$, then the max cluster size in $\mathcal{C}'$ is $6t\ell$. This satisfies condition 2 of being $\mathcal{C}$-good. In addition, it is fair by Proposition 34.

Finally, we utilize the fact that we only moved at most $\ell + 108t^2\ell^2$ vertices from any cluster, and note that we only move vertices of a certain color if we have $O(n)$ of that color in that cluster. Then by Lemma 38, we know we lost at most $O(\ell\gamma_t/t + t\gamma_t/\ell^2)$ fraction of the weight from any cluster. This satisfies the second condition and therefore $\mathcal{C}'$ is $\mathcal{C}$-good. □

Table 2.5: Impact of different fairlet decomposition on ratio over original average-linkage in percentage (mean $\pm$ std. dev).

Samples	100	200	400	800	1600
CensusGender, initial	74.12 ± 2.52	76.16 ± 3.42	74.15 ± 1.44	70.17 ± 1.01	65.02 ± 0.79
final	92.32 ± 2.70	95.75 ± 0.74	95.68 ± 0.96	96.61 ± 0.60	97.45 ± 0.19
CensusRace, initial	65.67 ± 7.53	65.31 ± 3.74	61.97 ± 2.50	59.59 ± 1.89	56.91 ± 0.82
final	85.38 ± 1.68	92.98 ± 1.89	94.99 ± 0.52	96.86 ± 0.85	97.24 ± 0.63
BankMarriage, initial	75.19 ± 2.53	73.58 ± 1.05	74.03 ± 1.33	73.68 ± 0.59	72.94 ± 0.63
final	93.88 ± 2.16	96.91 ± 0.99	96.82 ± 0.36	97.05 ± 0.71	97.81 ± 0.49
BankAge, initial	77.48 ± 1.45	78.28 ± 1.75	76.40 ± 1.65	75.95 ± 0.77	75.33 ± 0.28
final	91.26 ± 2.66	95.74 ± 2.17	96.45 ± 1.56	97.31 ± 1.94	97.84 ± 0.92

2.9.6 Additional experimental results for revenue

We have conducted experiments on the four datasets for revenue as well. The Table 2.5 shows the ratio of fair tree built by using average-linkage on different fairlet decompositions. We run Algorithm 1 on the subsamples with Euclidean distances. Then we convert distances into similarity scores using transformation $s(i,j) = \frac{1}{1+d(i,j)}$. We test the performance of the initial random fairlet decomposition and final fairlet decomposition found by Algorithm 1 for revenue objective using the converted similarity scores.

2.9.7 Additional experimental results for multiple colors

We ran experiments with multiple colors and the results are analogous to those in the paper. We tested both Census and Bank datasets, with age as the protected feature. For both datasets we set 4 ranges of age to get 4 colors and used $\alpha = 1/3$. We ran the fairlet decomposition in [Ahmadian et al., 2019] and compare the fair hierarchical clustering's performance to that of average-linkage. The age ranges and the number of data points belonging to each color are reported in Table 2.6. Colors are named $\{1, 2, 3, 4\}$ descending with regard to the number of points of the color. The vanilla average-linkage has been found to be unfair: if we take the layer

of clusters in the tree that is only one layer higher than the leaves, there is always one cluster with $\alpha > \frac{1}{3}$ for the definition of α-capped fairness, showing the tree to be unfair.

Table 2.6: Age ranges for all four colors for Census and Bank.

Dataset	Color 1	Color 2	Color 3	Color 4
CensusMultiColor	$(26, 38]$: 9796	$(38, 48]$: 7131	$(48, +\infty)$: 6822	$(0, 26]$: 6413
BankMultiColor	$(30, 38]$: 14845	$(38, 48]$: 12148	$(48, +\infty)$: 11188	$(0, 30]$: 7030

As in the main body, in Table 2.7, we show for each dataset the $\text{ratio}_{\text{value}}$ both at the time of initialization (Initial) and after using the local search algorithm (Final), where $\text{ratio}_{\text{value}}$ is the ratio between the performance of the tree built on top of the fairlets and that of the tree directly built by average-linkage.

Table 2.7: Impact of Algorithm 1 on $\text{ratio}_{\text{value}}$ in percentage (mean $\pm$ std. dev).

Samples	200	400	800	1600	3200	6400
CensusMultiColor, initial	88.55 ± 0.87	88.74 ± 0.46	88.45 ± 0.53	88.68 ± 0.22	88.56 ± 0.20	88.46 ± 0.30
final	99.01 ± 0.09	99.41 ± 0.57	99.87 ± 0.28	99.80 ± 0.27	100.00 ± 0.14	99.88 ± 0.30
BankMultiColor, initial	90.98 ± 1.17	91.22 ± 0.84	91.87 ± 0.32	91.70 ± 0.30	91.70 ± 0.18	91.69 ± 0.14
final	98.78 ± 0.22	99.34 ± 0.32	99.48 ± 0.16	99.71 ± 0.16	99.80 ± 0.08	99.84 ± 0.05

Table 2.8 shows the performance of trees built by average-linkage based on different fairlets, for Revenue objective. As in the main body, the similarity score between any two points i, j is $s(i, j) = \frac{1}{1+d(i,j)}$. The entries in the table are mean and standard deviation of ratios between the fair tree performance and the vanilla average-linkage tree performance. This ratio was calculated both at time of initialization (Initial) when the fairlets were randomly found, and after Algorithm 1 terminated (Final).

Table 2.8: Impact of Algorithm 1 on revenue, in percentage (mean $\pm$ std. dev).

Samples	200	400	800	1600	3200
CensusMultiColor, initial	75.76 ± 2.86	73.60 ± 1.77	69.77 ± 0.56	66.02 ± 0.95	61.94 ± 0.61
final	92.68 ± 0.97	94.66 ± 1.66	96.40 ± 0.61	97.09 ± 0.60	97.43 ± 0.77
BankMultiColor, initial	72.08 ± 0.98	70.96 ± 0.69	70.79 ± 0.72	70.77 ± 0.49	69.88 ± 0.53
final	94.99 ± 0.79	95.87 ± 2.07	97.19 ± 0.81	97.93 ± 0.59	98.43 ± 0.14

Table 2.9 shows the run time of Algorithm 1 with multiple colors.

Table 2.9: Average running time of Algorithm 1 in seconds.

Samples	200	400	800	1600	3200	6400
CENSUSMULTICOLOR	0.43	1.76	7.34	35.22	152.71	803.59
BANKMULTICOLOR	0.43	1.45	6.77	29.64	127.29	586.08

2.9.8 Pseudocode for the cost objective

The following pseudocode describes the process for achieving our fair cost approximation.

Algorithm 2 Fair hierarchical clustering for cost objective.

Graph G, edge weight $w : E \to \mathbb{R}$, color $c : V \to \{\text{red, blue}\}$, parameters t and ℓ

▷ Step (A)
$T \leftarrow \textsc{UnfairHC}(G, w)$ ▷ Blackbox unfair clustering that minimizes cost

▷ Step (B)
Let $\mathcal{C} \leftarrow \emptyset$
Do a BFS of T, placing visited cluster C in $\mathcal{C}$ if $|C| \leq t$, and not proceeding to C's children

▷ Step (C)
$\mathcal{C}_0, C' \leftarrow \emptyset$
for C **in** $\mathcal{C}$ **do**
    $C' \leftarrow C' \cup C$
    if $|C'| \geq t$ **then**
        Add C' to $\mathcal{C}_0$
        E Let $C' \leftarrow \emptyset$
    end if
end for
If $|C'| > 0$, merge C' into some cluster in $\mathcal{C}_0$

▷ Step (D)
for C **in** $\mathcal{C}_0$ **do**
    Let $exc(C) \leftarrow$ majority color in C
    Let $ex(C) \leftarrow$ difference between majority and minority colors in C
end for

▷ Step (E)
$H_M \leftarrow$ BuildClusteringGraph$(\mathcal{C}_0, ex, exc)$

▷ Step (F)
$fV \leftarrow$ FixUnmatchedVertices$(\mathcal{C}_0, H_M, ex, exc)$

▷ Step (G)
$\mathcal{C}' \leftarrow$ ConstructClustering$(\mathcal{C}_0, ex, exc, fV)$
Return $\mathcal{C}'$

Algorithm 3 BuildClusteringGraph $(\mathcal{C}_0, ex, exc)$

$H_M \leftarrow (V_M = \mathcal{C}_0, E_M = \emptyset)$
Let $C_i \in V_M$ be any vertex
Let $\ell \leftarrow n^{1/3}\sqrt{\log n}$
while $\exists$ an unvisited $C_j \in V_M$ such that $exc(C_j) \neq exc(C_i)$ **do**
    Add (C_i, C_j) to E_M
    Swap labels C_i and C_j if $ex(C_j) > ex(C_i)$
    Let $ex(C_i) \leftarrow ex(C_i) - ex(C_j)$
    if $ex(C_i) < \ell$ or $|component(C_i)| \geq \ell$ **then**
        Reassign starting point C_i to an unvisited vertex in V_M
    end if
end while
Return H_M

Algorithm 4 FixUnmatchedVertices$(\mathcal{C}_0, H_M, ex, exc)$

Let $\ell \leftarrow n^{1/3}\sqrt{\log n}$
for $C \in \mathcal{C}_0 \setminus V_M$ **do**
    Let $fV(C, \text{red}), fV(C, \text{blue}) \leftarrow m^2/\ell^2$
end for
for i **from** 1 **to** $108t^2/\ell^3$ **do**
    for each k component in H_M **do**
        for p **in** a BFS of k **do**
            Let $ch \leftarrow$ some child of p
            $fV(p, exc(p)) \leftarrow fV(p, exc(p)) + \ell$
            $ex(p) \leftarrow ex(p) - \ell$
            $fV(ch, exc(ch)) \leftarrow fV(ch, exc(ch)) + \ell$
            $ex(ch) \leftarrow ex(ch) - \ell$
            if # matches between p and $ch < \ell$ **then**
                Remove (p, ch) from E_M

                ▷ This creates a new component
            end if
        end for
    end for
end for
Return fV

Algorithm 5 ConstructClustering($\mathcal{C}_0, ex, exc, fV$)

Let $\mathcal{C}', R \leftarrow \emptyset$
for C in $\mathcal{C}_0$ **do**
    for c **in** {red, blue} **do**
        Let $f = fV(C, c)$
        Let $C_f = \{v \in C : c(v) = c\}$
        Create the transformed graph L from C_f ▷ Described in the proof of Lemma 38
        $C' \leftarrow$ MINWEIGHTBISECTION(L) ▷ Blackbox, returns isolated C_f vertices
        $C \leftarrow C \setminus C'$
        $R \leftarrow R \cup C'$
        $ex(C) \leftarrow ex(C) - |C'|$
    end for
end for
for $C \in \mathcal{C}_0$ **do**
    Let $S \subset R$ such that $|S| = ex(C)$ with no vertices of color $exc(C)$
    $C = C \cup S$
    $R \leftarrow R \setminus S$
    Add C to $\mathcal{C}'$
end for
Return $\mathcal{C}'$

Chapter 3: Generalized Reductions: Making any Hierarchical Clustering Fair and Balanced with Low Cost

3.1 Introduction

Fair machine learning, and namely clustering, has seen a recent surge as researchers recognize its practical importance. In spite of the clear and serious impact the lack of fairness in existing intelligent systems has on society [Angwin et al., 2016, Rieke and Bogen, 2018, Ledford, 2019, Sweeney, 2013], and despite significant progress towards fair flat (not hierarchical) clustering [Ahmadian et al., 2020c, Backurs et al., 2019, Bera et al., 2019, Brubach et al., 2020, Chakrabarti et al., 2021, Chen et al., 2019, Chierichetti et al., 2017, Esmaeili et al., 2021, Esmaeili et al., 2020, Kleindessner et al., 2019a, Rösner and Schmidt, 2018], fairness in hierarchical clustering has only received some recent attention [Ahmadian et al., 2020b, Chhabra and Mohapatra, 2022]. Thus, we are some of the first to study this problem.

Hierarchical clustering (Figure 3.1) is the well-known extension to clustering, where we create a hierarchy of subclusters contained within superclusters. This structure forms a tree (a

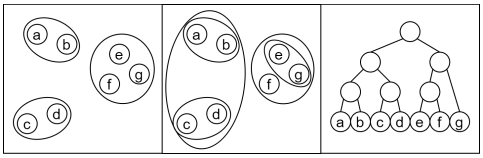


Figure 3.1: On the left is a 3-clustering, in the center is a hierarchical clustering, and on the right is its dendrogram.

dendrogram), where leaves represent the input data. An internal node v corresponds to the cluster of all the leaves of the subtree rooted at v. The root is the cluster of all data.

Hierarchical clusterings more completely illustrate data relationships than flat clusterings. For instance, they are commonly used in phylogenetics to depict the entire evolutionary history of species, whereas a clustering would only depict species similarities. It also has a myriad of other uses in machine learning applications such as search [Cai et al., 2004, Ferragina and Gulli, 2005, Kou and Lou, 2012], social network analysis [Leskovec et al., 2014, Mann et al., 2008], and image recognition [Arifin and Asano, 2006, Lin et al., 2018, Pan et al., 2016]. On top of this, hierarchical clusterings can also be used to solve flat clustering when the number of clusters is not given. To do this, we extract clusterings at different resolutions in the hierarchy that all "agree" (if two points are together in a cluster, then they will also be together in any larger cluster) and select the one that best fits the application.

Hierarchical clusterings can be evaluated using a number of metrics. Perhaps most notably, [Dasgupta, 2016] introduced cost (Definition 40), an intuitive and explainable metric which exhibits numerous desirable properties and has become quite popular and well-respected [Charikar and Chatziafratis, 2017, Charikar et al., 2019b, Chatziafratis et al., 2018, Cohen-Addad et al., 2017, Roy and Pokutta, 2016]. Unfortunately, it is difficult to approximate, where the best existing solutions require semi-definite programs [Dasgupta, 2016, Charikar and Chatziafratis, 2017], and it is not efficiently $O(1)$-approximable by the Small-Set Expansion Hypothesis [Charikar and Chatziafratis, 2017]. The $\mathrm{revenue}$ [Moseley and Wang, 2017] and value [Cohen-Addad et al., 2018] metrics, both derived from cost, exhibit $O(1)$-approximability, but are not as explainable or appreciated.

Only two papers have explored fair hierarchical clustering [Ahmadian et al., 2020b, Chhabra

and Mohapatra, 2022]. Both extend fairness constraints from fair clustering literature that trace back to [Chierichetti et al., 2017]'s *disparate impact*. Consider a graph $G = (V, E, w)$, where each point has a color, which represents a protected class (e.g., gender, race, etc.). On two colors, red and blue, they consider a clustering fair if the ratio between red and blue points in each cluster is equal to that in the input data. This ensures that the impact of a cluster on a protected class is proportionate to the class size. The constraint has been further generalized [Ahmadian et al., 2019, Bercea et al., 2019]: given a dataset with λ colors and constraint vectors $\vec{\alpha}, \vec{\beta} \in (0, 1)^{\lambda}$, a clustering is fair if for all $\ell \in [\lambda]$ and every cluster C, $\alpha_\ell |C| \leq \ell(C) \leq \beta_\ell |C|$, were $\ell(C)$ is the number of points in C of color ℓ. Naturally, then, a hierarchical clustering is fair if every non-singleton cluster in the hierarchy satisfies this constraint (with nuances, see Section 3.2.2), as in [Ahmadian et al., 2020b].

This work explores broad guarantees, namely cost approximations, for fair hierarchical clustering. The only previous algorithm is quite complicated and only yields a $O(n^{5/6} \log^{5/4} n)$ fair approximation for cost (where an $O(n)$-approximation is trivial) [Ahmadian et al., 2020b], and it assumes two, equally represented colors. This reflects the inherent difficulty of finding solutions that are low-cost as opposed to high-$\mathrm{revenue}$ or high-value, both of which exhibit fair $O(1)$-approximations [Ahmadian et al., 2020b]). Our algorithms improve previous work in quite a few ways: 1) we achieve a near-exponential improvement in approximation factor, 2) our algorithm works on $O(1)$ instead of only 2 colors, 3) our work handles different representational proportions across colors in the initial dataset, 4) we simultaneously guarantee fairness and relative cluster balance, and 5) our methods, which modify a given (unfair) hierarchy, have measurable, explainable, and limited impacts on the structure of the input hierarchy.

	Achieves	Approximation	Fairness	Colors	Color ratios	Explained
Past	Determ. fairness	$O(n^{5/6}\text{polylog } n)$	Perfect	2	50/50 only	No
This Work	Determ. fairness	$O(n^{\delta}\text{polylog } n)$	Approx	$O(1)$	$O(1)$	Yes
	Stoch. fairness	$O(\log^{3/2} n)$	Approx	$O(1)$	$O(1)$	Yes
	ϵ-rel. balance	$O(\sqrt{\log n}/\epsilon)$	N/A	N/A	N/A	Yes
	$1/6$-rel. balance	$O(\sqrt{\log n})$	N/A	N/A	N/A	Yes

Table 3.1: Our versus previous work. Note $\delta \in (0, 1/6)$ is parameterizable, trading approximation factor for fairness. Our algorithms are explainable in that the alterations made to the hierarchy are clear and well-defined.

3.1.1 Our Contributions

This work proposes new algorithms for fair and balanced hierarchical clustering. A summary of our work can be found in Table 3.1.

We introduce four simple hierarchy tree operators which have clear, measurable impacts. We show how to compose them together on a (potentially unbalanced and unfair) hierarchy to yield a fair and/or balanced hierarchy with similar structure. This process clarifies the functionality of our algorithms and illustrates the modifications imposed on the hierarchy. Each of our four proposed algorithms starts with a given γ-approximate (unfair) hierarchical clustering algorithm (i.e., [Dasgupta, 2016]'s $O(\sqrt{\log n})$-approximation) and then builds on top of each other, imposing a new operator to achieve a more advanced result. Additionally, each algorithm stands alone as a unique contribution.

Our first algorithm produces a $1/6$-relatively balanced hierarchy that $\frac{3}{2}\gamma$-approximates cost (see Theorem 50).[1] Here, ϵ-relative balance means that at each split in the hierarchy, a cluster splits in half within a proportional error of up to $1+\epsilon$ (see Definition 43). Starting at the root, the

[1]Repeated sparsest cuts achieves this with similar cost. Our algorithms, though, can be used explainably on top of existing unfair algorithms and may perform better as unfair research progresses.

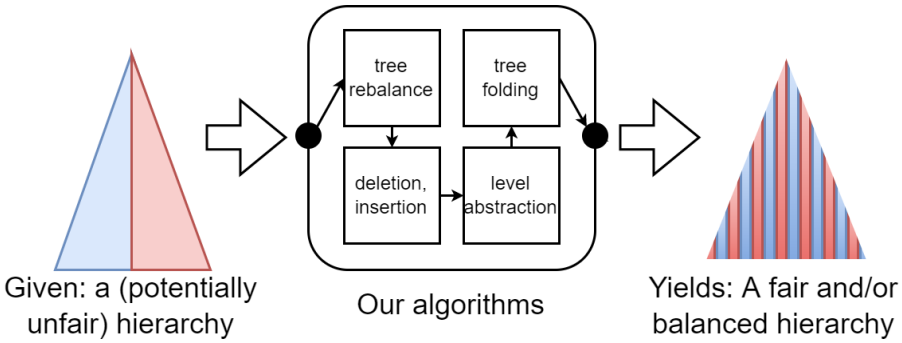


Figure 3.2: Our algorithms take a potentially unfair hierarchical clustering, apply our tree operators, and yield fair and/or balanced hierarchies.

algorithm recursively applies our *tree rebalance* (see Definition 46) operator. This restructures the tree by moving some subtree up to become a child of the root. It preserves much of the hierarchy's structure while achieving relative balance.

Our next result refines this to achieve ϵ-relative balance for any $\epsilon \in (0, 1/6)$ that $\frac{9}{2\epsilon}\gamma$-approximates cost (see Theorem 54). This can get arbitrarily close to creating a perfectly balanced hierarchy. To achieve this, we simply run our first algorithm and then apply a limited number of *subtree deletion and insertion operators* (see Definition 47). This operator selects a subtree, removes it, and reinserts elsewhere. It again preserves much of T's structure.

Third, we propose an algorithm for stochastically fair hierarchical clustering (see Definition 42). Under certain stochastic parameterizations and arbitrarily many colors, the algorithm achieves stochastic fairness and $O(\gamma \log n)$-approximates cost (see Theorem 58). This is quite impressive, as the best previous fair approximation (albeit, for deterministic colors) was $poly(n)$ [Ahmadian et al., 2020b]. To achieve this novel result, we first find an $O(1/\log n)$-relatively balanced hierarchy and then apply our level abstraction operator once to the bottom layers of the hierarchy. This operator removes selected layers, setting much lower descendants of a vertex as direct children. While this removes details in the hierarchy, the remaining structure still agrees with the original tree. This simple addition guarantees fairness under stochastic color

assignment.

Our main result finds an approximately fair hierarchical clustering that $O(n^{\delta} poly \log n)$-approximates cost (see Theorem 63), where $\delta \in (0, 1)$ is a given constant. This is a near-exponential improvement over previous work which only achieves a $O(n^{5/6} poly \log(n))$ approximation on two equally represented colors. On top of that, our algorithm works on many colors, with many different color ratios, and achieves a simultaneously balanced hierarchy in an explainable manner. The algorithm, FairHC (Algorithm 8), builds on top of our stochastic algorithm (parameterized slightly differently, see Section 3.4.4) before applying a new operator: tree folding. Tree folding maps isomorphic trees on top of each other. In hierarchical clustering, this means taking two subtrees, mapping clusters in one tree to the other, and then merging clusters according to the mapping. Matching up clusters with different proportions of colors helps balance out the color ratios across clusters, which gives us our fairness result.

3.2 Preliminaries

Our input is a complete weighted graph $G = (V, E, w)$ where $w : E \to \mathbb{R}^+$ is a similarity measure. A hierarchical clustering can be defined as a hierarchy tree T, where its leaves are $\mathrm{leaves}(T) = V$, and internal nodes represent the merging of vertices into clusters and clusters into superclusters.

3.2.1 Optimization Problem

We use [Dasgupta, 2016]'s cost function as an optimization metric. For simplicity, we let $n_T(e)$ denote the size of smallest cluster in T containing both endpoints of e. In other words,

for $e = (u, v)$, $n_T(e) = |\text{leaves}(T[u \wedge v])|$, where $u \wedge v$ is the lowest common ancestor of u and v in T and $T[u]$ for any vertex u is the subtree rooted at u. We additionally denote $n_T(u) = |\text{leaves}(T[u])|$ for internal node u. Also we let $\text{root}(T)$ be the root of T, and $\text{left}_T(u)$ and $\text{right}_T(u)$ access left and right children respectively. We can evaluate the cost contribution of an edge to a hierarchy.

Definition 39. *The **cost** of $e \in E$ in a graph $G = (V, E, w)$ in a hierarchy T is $\text{cost}_T(e) = w(e) \cdot n_T(e)$.*

We strive to minimize the sum of costs across all edges.

Definition 40 ([Dasgupta, 2016])**.** *The **cost** of a hierarchy T on graph $G = (V, E, w)$ is:*

$$\text{cost}(T) = \sum_{e \in E} \text{cost}_T(e)$$

[Dasgupta, 2016] showed that we can assume that all unfair trees optimizing for cost are binary. Note that we must consider non-binary trees when we incorporate fairness as it may not allow binary splits at its lowest levels.

3.2.2 Fairness and Stochastic Fairness

We consider the fairness constraints based off those introduced by [Chierichetti et al., 2017] and extended by [Bercea et al., 2019]. On a graph G with colored vertices, let $\ell(C)$ count the number of ℓ-colored points in cluster C.

Definition 41. *Consider a graph $G = (V, E, w)$ with vertices colored one of λ colors, and two vectors of parameters $\alpha, \beta \in (0, 1)^\lambda$ with $\alpha_\ell \leq \beta_\ell$ for all $\ell \in [\lambda]$. A hierarchy T on G is **fair** if*

for any non-singleton cluster C in T and for every $\ell \in [\lambda]$, $\alpha_\ell |C| \leq \ell(C) \leq \beta_\ell |C|$. Additionally, any cluster with a leaf child has only leaf children.

Notice that the final constraint regarding leaf-children simply enforces that a hierarchy must have some "baseline" fair clustering (e.g., a fairlet decomposition [Chierichetti et al., 2017]). Consider a tree that is just a stick with individual leaf children at each depth. While internal nodes may represent fair clusters, you cannot extract any nontrivial fair flat clustering from this, since it must contain a singleton, which is unfair. We view such a hierarchy This is clearly undesirable, and our additional constraint prevents this issue.

In the stochastic problem, points are assigned colors at random. We must ensure that with high probability (i.e., at least $1 - 1/\text{polylog}\,(n)$) all clusters are fair.

Definition 42. *Consider the same context as Definition 41 with an additional function $p_\ell : v \to (0, 1)$ denoting the probability v has color ℓ such that $\sum_{\ell=1}^{\lambda} p_\ell(v) = 1$ and each vertex has exactly one color. An algorithm is **stochastically fair** if, with high probability, it outputs a fair hierarchy.*

3.3 Tree Properties and Operators

This work is interested in both fair and balanced hierarchies. Balanced trees have numerous practical uses, and in this paper, we show how to use them to guarantee fairness too.

Definition 43. *A hierarchy T is ϵ-**relatively balanced** if for every pair of clusters C and C' that share a parent cluster C_p with $|C_p| \geq 1/(2\epsilon)$ in T, $(1/2 - \epsilon)|C_p| \leq |C|, |C'| \leq (1/2 + \epsilon)|C_p|$.*

Notice that we only care about splitting clusters C_p with size satisfying $|C_p| \geq 1/(2\epsilon)$. This is because, on smaller clusters, it may be impossible to divide them with relative balance. For

instance, if $|C_p| = 3$, we know we can only split it into a 1-sized and 2-sized cluster, yielding a minimum relative balance of $1/6$. For smaller ϵ, we require larger cluster sizes to make this possible.

We will often discuss the "separation" of edges in our proposed operators. It refers to occasions when a point is added to the first cluster that contains both endpoints. We do not care if points are removed.

Definition 44. *An edge $e = (u, v)$ is (or its endpoints are) **separated** by an operator which changes hierarchy T to T' if* $\text{cluster}_T(u \wedge v) \not\supseteq \text{cluster}_{T'}(u \wedge v)$.

Almost definitionally, if an edge is not separated by an operator, then the cluster size at its lowest common ancestor does not increase. Thus, its cost contribution does not increase.

3.3.1 Tree Operators

Our work uses a number of different tree operations to modify and combine trees (Figure 3.3). These illustrate exactly how our algorithms alter the input. We show how many operators of each type each of our algorithms use and to what extent they affect the hierarchy through a metric we propose here. Notably, for each proposed algorithm on an input T, it transforms T into output T' by *only applying our four tree operators: tree rebalance, subtree deletion and insertion, level abstraction, and subtree folding.*

Each operator has an associated operation cost, which measures the proportional increase in cost of each edge separated by the operation. We present lemmas that bound the operation cost of each operator in the Appendix.

Definition 45. *Assume we apply some tree operation to transform T into T'. The **operation***

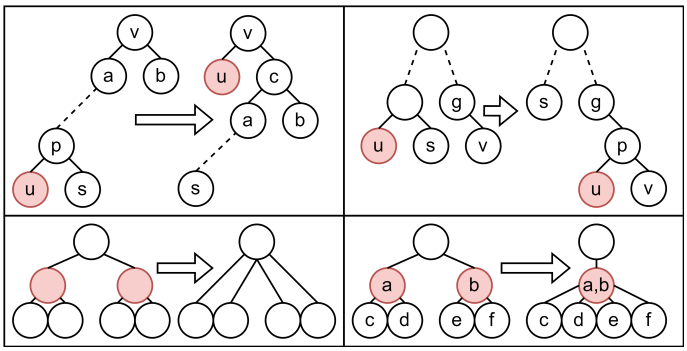


Figure 3.3: We depict our tree operators: tree rebalance (top left), subtree deletion and insertion (top right), level abstraction (bottom left), and tree folding (bottom right).

cost *is an upper bound* Δ *such that for any edge* e *that is separated by the operation, then* $\text{cost}_{T'}(e) \leq \Delta\, \text{cost}_T(e)$.

The first operation is a tree rebalance, which rotates in a descendant of the root to instead be a direct child. This defines our first result in Theorem 50, as clever use of the tree rebalance operator allows us to find a relatively balanced tree. This is illustrated in the top left panel of Figure 3.3.

Definition 46. *Consider a binary tree* T *with internal nodes* v*,* v*'s descendant* u*, and* v*'s children* a *and* b*. A* ***tree rebalance*** *of* u *at* v *(*$\text{tree\_rebalance}(u, v)$*) puts a new node* c *in between* v *and sibling nodes* a *and* b*. It then removes* $T[u]$ *from* $T[a]$ *and sets* u *to be* v*'s other child.*

Tree rebalancing will only yield $1/6$-relatively balanced trees, which is interestingly something [Dasgupta, 2016]'s sparsest cut algorithm, one of the current best cost approximations, achieves via a similar analysis. To refine this, we use subtree insertion and deletion (Figure 3.3, top right). At a root with large child a and small child b, we can move a small subtree from a to b to improve the balance.

Definition 47. *Consider a binary tree* T *with internal nodes* u*, some non-ancestor* v*,* u*'s sibling* s*, and* v*'s parent* g*.* ***Subtree deletion*** *at* u *removes* $T[u]$ *from* T *and contracts* s *into its parent.*

Subtree insertion *of $T[u]$ at v inserts a new parent p between v and g and adds u as a second child of p. The operator* $\mathrm{del\_ins}(u, v)$ *deletes u and inserts $T[u]$ at v.*

We will also need to abstract away (Figure 3.3, bottom left) certain levels of the hierarchy to simplify it. This involves taking vertices at depth d_2 and iteratively merging them into their parents until they reach depth d_1. In other words, ignore tree structure between two levels of the hierarchy.

Definition 48. *Consider a binary tree T with two parameters d_1 and d_2 such that $d_1 < d_2 <$* $\mathrm{height}(T)$. ***Level abstraction*** *between levels $d_1 + 1$ and d_2 (*$\mathrm{abstract}(d_1, d_2)$*) involves taking all internal nodes between depths $d_1 + 1$ and d_2 in T and contracting them into their parents.*

To achieve fairness in Section 3.4, we use tree folding (Figure 3.3, bottom right). Given multiple isomorphic trees (ignoring leaves), we map the topologies of the trees together.

Definition 49. *Consider a set of subtrees $T_1, \ldots, T_k$ of T such that all trees T_i without their leaves have the same topology, and all* $\mathrm{root}(T_i)$ *have the same parent p in T. This means that for each $i \in [k]$, there is a tree isomorphism $\phi_i : I_i \to I_k$ where I_i and I_k are the internal nodes of the corresponding trees. A* ***tree folding*** *of trees $T_1, \ldots, T_k$ (*$\mathrm{fold}(T_1, \ldots, T_k)$*) modifies T such that all $T_1, \ldots, T_k$ are replaced by a single tree T_f whose* $\mathrm{root}(T)$ *is made a child of p and T_f has the same internal topology as I_k such that for any leaf ℓ of any tree T_i with parent p_i in T_i, we set its parent to $\phi_i(p_i)$.*

3.4 Fair and Balanced Reductions

We now present our main algorithms, which sequentially build on top of each other, adding new operators in a limited, measureable capacity to achieve new results.

3.4.1 Relatively Rebalanced Trees

Our first algorithm guarantees $1/6$-relative balance. It only modifies the tree through a series of limited tree rebalances (Definition 46). We can show that this only incurs a small constant factor proportionate increase in cost.

Theorem 50. *Given a γ-approximation for cost, we can construct a $\frac{3}{2}\gamma$-approximation for cost which guarantees $\frac{1}{6}$-relative balance. It only modifies the tree by applying tree rebalance operators of operation cost $3/2$, and every edge is only separated by at most one such operator.*

Algorithm 6 RebalanceTree

Input A hierarchy tree T of size n, with smaller cluster always on the left.
Output A $\frac{1}{6}$-rebalanced T-tree.

$r, v = \text{root}(T)$
$A = \text{leaves}(\text{left}_T(v))$
while $|A| \geq \frac{2}{3}n$ **do**
    $v \leftarrow \text{left}_T(v)$
    $A \leftarrow \text{leaves}(\text{left}_T(v))$
end while
$T \leftarrow T.\text{tree\_rebalance}(v, r)$
Let $T'_l = \text{RebalanceTree}(T[\text{left}_T(r)])$
Let $T'_r = \text{RebalanceTree}(T[\text{right}_T(r)])$
Return T' with root r with $\text{left}(r) = \text{root}(T'_l)$ and $\text{right}(r) = \text{root}(T'_r)$

This idea was first introduced by [Dasgupta, 2016] as an analytical tool for their algorithm. However we use it more explicitly here to take any given hierarchy and rearrange it to be balanced. The basic idea is to start with some given tree T. Draw T from top to bottom such that the smaller cluster in a split is put on the left. Let A_1 and B_1 be our first split. Continue working on the left side, splitting A_1 into A_2 and B_2 and so on. Stop when we find the first cluster B_k such that $|B_k| \geq n/3$. This defines our first split: partition V into A_k and $B = \cup_{i=1}^{k} B_i$. Then recurse on each side.

It is not too hard to see that this yields a $\frac{1}{6}$-relatively balanced tree. Our search stopping conditions enforce this.

Lemma 51. *Algorithm 6 produces a $\frac{1}{6}$-relatively balanced tree.*

The next property also comes from the fact that once an edge is separated, it will never be separated again.

Lemma 52. *In Algorithm 6, every edge is separated by at most one tree rebalance operator.*

Finally, the operation cost of the rebalance operators comes from our stopping threshold.

Lemma 53. *In Algorithm 6, every tree rebalance operator has operation cost at most $3/2$.*

In Theorem 50, the relative balance comes from Lemma 51, the operator properties come from Lemmas 52 and 53 respectively, and the approximation factor comes from Lemma 52 and Lemma 53 together.

3.4.2 Refining Relatively Rebalanced Trees

We now propose a significant extension of Algorithm 6 which allows us to get a stronger balance guarantee. Specifically (where ϵ may be a function of n):

Theorem 54. *Given a γ-approximation for cost, we can construct a $\frac{9\gamma}{2\epsilon}$-approximation for cost which guarantees ϵ-relative balance for $0 < \epsilon \leq 1/6$. In addtition to Theorem 50, it only modifies the tree by applying subtree deletion and insertion operators of operation cost $\frac{3}{\epsilon}$, and every edge is only separated by at most one such operator.*

To do this, we first apply $\mathrm{RebalanceTree}$. Then, at each split starting at the root, we execute $\mathrm{SubtreeSearch}$ (Algorithm 9 in Appendix 3.8.2), which searches for a small subtree below the

right child, deletes it, and moves it below the left child in order to reduce the error in the relative balance. If we do this enough, we can reduce the relative balance to ϵ. We call our algorithm, in Algorithm 7, RefineRebalanceTree.

Algorithm 7 RefineRebalanceTree

Input A $\frac{1}{6}$-relatively balanced hierarchy tree T of size n, with smaller cluster always on the left, and balance parameter $\epsilon \in (0, 1/6)$.

Output An ϵ-relatively balanced tree.

if $\epsilon \leq 1/(2n)$ **then**
    Return T
end if
$v = \text{root}(T)$
Let $T_{big} = T[\text{left}_T(v)]$, $T_{small} = T[\text{right}_T(v)]$
while $|\text{leaves}(T_{big})| \geq (1/2 + \epsilon)n$ **do**
    $\delta \leftarrow (|\text{leaves}(T_{big})| - n/2)/n$
    Let $T_{big} = \text{SubtreeSearch}(T_{big}, \delta n)$
end while
$T_{big} \leftarrow \text{RefineRebalanceTree}(T_{big}, \epsilon)$
$T_{small} \leftarrow \text{RefineRebalanceTree}(T_{small}, \epsilon)$
Return T' with root r with $\text{left}(r) = \text{root}(T_{big})$ and $\text{right}(r) = \text{root}(T_{small})$

We can show that this algorithm creates a nicely rebalanced tree. SubtreeSearch specifically guarantees a proportional $2/3$ reduction in relative balance (see Appendix 3.8.2). Therefore, enough executions of SubtreeSearch will make the split ϵ-relatively balanced, and recursing down the tree guarantees that the entire tree is ϵ-relatively balanced.

Lemma 55. *Algorithm 7 produces an ϵ-rebalanced tree.*

To bound the operators on top of those used by Algorithm 6, note that we only apply the subtree deletion and insertion operators. Additionally, each edge cannot be separated more than once.

Lemma 56. *In Algorithm 7, every edge is separated by at most one subtree deletion and insertion operator.*

Finally, we can also limit the operation cost of the subtree deletion and insertion operator. This is because we limit the depth of the $\mathrm{SubtreeSearch}$ function as it will never be given a parameter below ϵn.

Lemma 57. *In Algorithm 7, every subtree deletion and insertion operator has operation cost at most* $3/\epsilon$.

For Theorem 54, the relative balance comes from Lemma 55, the operator properties com from Lemmas 56 and 57 respectively, and the approximation factor comes from Lemma 69, Lemma 56, and Lemma 57 together.

3.4.3 Stochastically Fair Hierarchical Clustering

At this point, we almost have enough tools to solve stochastically fair hierarchical clustering. For this, however, we need a simple application of level abstraction (Definition 48). We introduce $\mathrm{StochasticallyFairHC}$, which simply imposes one level abstraction: $T' = T.\mathrm{abstract}(t, h_{max})$ on the bottom levels of the hierarchy. Here, t is a parameter and h_{max} is the max depth in T. Notice that we require the input to be relatively balanced to achieve this result.

Theorem 58. *Given a* γ*-approximation for* cost *and any* $\epsilon = 1/(c\log_2 n)$, $c, \lambda = O(1)$, *and* $\delta \in (0,1)$, *in the stochastic fairness setting with* $\frac{1}{1-\delta}\alpha_\ell \le p_\ell(v) \le \frac{1}{1+\delta}\beta_\ell$ *for all* $v \in V$ *and* $\ell \in [\lambda]$, *there is a* $e^{4/(c(1-o(1))} \cdot \frac{3(1-\delta)\ln(cn)}{\delta^2 \min_{\ell\in[\lambda]}\alpha_\ell} \cdot \frac{9\gamma}{2\epsilon}$ *fair approximation that succeeds with high probability. On top of the operators of Theorem 54, it only modifies the tree by applying one level abstraction of operation cost at most* $e^{4/(c(1-o(1))} \cdot \frac{3\ln(cn)}{a\delta^2}$.

Theorem 58 with constant α_ℓ for all $\ell \in [\lambda]$, c, and δ yields a $O(\gamma \log n)$ approximation. Since $\gamma = O(\sqrt{\log n})$ [Charikar and Chatziafratis, 2017, Dasgupta, 2016], this becomes

$O(\log^{3/2} n)$. It is quite impressive, as the best previous fair (albeit, deterministic) approximation was $poly(n)$ [Ahmadian et al., 2020b]. Also, δ exhibits an important tradeoff: increasing δ increases success probability but also decreases the range of acceptable $p_\ell(v)$ values.

It might be tempting to suggest applying StochasticallyFairHC to any existing hierarchy, as opposed to one that is ϵ-relatively balanced. However, if we consider, for instance, a highly unbalanced tree where all internal nodes have at least one leaf-child, the algorithm would only merge the bottom t internal nodes into a single cluster, thereby not guaranteeing fairness. The resulting structure would also not be particularly interesting. This is why the rebalancing process is important.

Obviously, since we only apply level abstraction once, edge separation only happens once per edge in the algorithm. To bound the operation cost, we explore the relative size of clusters at a specified depth in the hierarchy. The following guarantee is achieved by considering a root-to-vertex path where we always travel to maximally/minimally sized clusters according to the tree's relative balance.

Lemma 59. *Let T be an ϵ-relatively balanced tree and u and v be internal nodes at depth i in T. Then $(1/2-\epsilon)^i n \leq n_T(u), n_T(v) \leq (1/2+\epsilon)^i n$, which also implies $\frac{n_T(u)}{n_T(v)} \leq \frac{(1+2\epsilon)^i}{(1-2\epsilon)^i}$. Additionally, if $i \leq \log_{1/2-\epsilon}(x/n)$ for some arbitrary $x \geq 1$ and $\epsilon = 1/(c\log_2 n)$ for a constant c, then the maximum cluster size is at most $e^{4/(c(1-o(1)))}x$.*

This yields our operation cost, since it bounds the size of clusters at certain depths.

Lemma 60. *In* StochasticallyFairHC*, the level abstraction has operation cost at most $(1/2+\epsilon)^t n$.*

To get our fairness results, we need to use a Chernoff bound, thus we must guarantee that

all internal nodes have sufficiently large size. This too comes from our bounds on cluster sizes.

Lemma 61. *In* StochasticallyFairHC*, for any internal node* v*,* $n_{T'}(v) \geq (1/2 - \epsilon)^t n$*.*

Finally, we must show the fairness guarantee. Since the union of two fair clusters is fair, we only need to show this for the clusters at height 1 in the hierarchy, as this would imply fairness for the rest of the hierarchy. This comes from a Chernoff bound.

Lemma 62. *The resulting tree from* StochasticallyFairHC *with* $t = \log_{1/2-\epsilon}\left(\frac{3(1-\delta)\ln(cn)}{a\delta^2 n}\right)$ *for* $a = \min_{\ell\in[\lambda]}\alpha_\ell$ *and any* $\delta > 0$ *is stochastically fair for given parameters* α_ℓ, β_ℓ *for all colors* $\ell \in [\lambda]$ *with high probability if with* $\frac{1}{1-\delta}\alpha_\ell \leq p_\ell(v) \leq \frac{1}{1+\delta}\beta_\ell$ *for all* $v \in V$ *and* $\ell \in [\lambda]$ *for* $\lambda = O(1)$*.*

This is sufficient to show Theorem 58. The fairness is a result of Lemma 62, the operator properties are a result of Lemma 60 and the obvious fact that we only apply one level abstraction, and the approximation factor comes from Lemma 70 and Lemma 60 together.

3.4.4 Deterministically Fair Hierarchical Clustering

Finally, we have our main results on the standard, deterministic fair hierarchical clustering problem. This algorithm builds on top of the results from Theorem 54 and uses methods similar to Theorem 58. In addition to previous algorithms, it uses more applications of level abstraction and introduces tree folding.

Theorem 63. *Given a* γ*-approximation for* cost *over* $\ell(V) = c_\ell n = O(n)$ *vertices of each color* $\ell \in [\lambda]$ *with* $h = n^\delta$ *for any constants* c, δ, k*, there is an algorithm that yields a hierarchy* T' *that:*

1. *Is a* $e^{\frac{4\log_2 n}{c(1-o(1))\lambda\log_2 h} + \frac{2}{c} + \frac{4}{c(1-o(1))}} \cdot \frac{9c^2\gamma}{4} \cdot n^\delta \log_2^2 n$*-approximation for cost.*

2. *Is fair for any parameters for all* $\ell \in [\lambda]$*:* $\beta_\ell \geq c_\ell \left(\frac{e^{4/c}}{kc_\ell} + e^{6/c}\right)^{1/\delta}$ *and* $\alpha_\ell \leq \frac{c_\ell}{e^{(6/c)\log_h(n)}}$.

On top of the operators of Theorem 58, it only modifies the tree by applying level abstraction of operation cost at most $e^{2/c}n^\delta$ *and tree folding of operation cost* $ke^{4/(c(1-o(1))}$ *on* k *subtrees, and each edge is separated in at most one level abstraction operator and in at most* λ/δ *tree fold operators.*

This algorithm runs in $O(n^2 \log n)$ *time.*

Since $\gamma = O(\sqrt{\log n})$, this becomes $O(n^\delta \log^{5/2} n)$ for any constant c, $\delta \in (0, 1)$, and k which greatly improves the previous $O(n^{5/6} \log^{5/4}(n))$-approximation [Ahmadian et al., 2020b]. Additionally, the previous work only considered 2 colors with equal representation in the dataset. Our algorithm greatly generalizes this to both more colors and different proportions of representation. While we do not guarantee exact color ratio preservation as the previous work does, our algorithms can get arbitrarily close through parameterization and we no longer require the ratio between colors points in the input to be exactly 1.

In terms of fairness, all of the variables here are parameterizeable constants. Increasing k, c, and δ will all make these values get closer to the true proportions of the colors in the overall dataset, and this can be done to an arbitrary extent. Therefore, based off the parameterization, this allows us to enforce clusters to have pretty close to the same color proportions as the underlying dataset.

The goal of this algorithm is to recursively abstract away the top $\log_2 h$ depth of the tree, where we end up setting $h = n^\delta$. Each time we do this, we get a kind of "frontier clustering", which is an h-sized clustering whose parents in the tree are all the root after level abstraction. Since the subtrees rooted at each cluster have the same topology (besides their leaves, this is

due to our level abstraction at the lowest levels in the tree), we can then execute tree folding on any subset of them. We select cluster subtrees to fold together such that, once we merge the appropriate clusters, the clustering at this level will be more fair. Then, as we recurse down the tree, we subsequently either eliminate clusters (via level abstraction) or fold them to guarantee fairness. For more information, see Algorithm 8.

Algorithm 8 FairHC

Input An $\epsilon = 1/(c \log_2 n)$ relatively balanced hierarchy tree T of size n on red and blue points, and parameters $h = 2^i$ and $k = 2^j$ for some $0 < j < i < \log_{1/2-\epsilon}(1/(2n\epsilon))$

Output A fair tree.

Let $T \leftarrow T.\text{abstract}(0, i)$
if T is height 1 **then**
    Return T
end if
Let $\mathcal{V}$ be the children of $\text{root}(T)$
for each color $\ell \in [\lambda]$ **do**
    Order $\mathcal{V} = \{v_i\}_{i \in [h]}$ decreasing by $\frac{\ell(\text{leaves}(v_i))}{|\text{leaves}(v_i)|}$
    For all $i \in [k]$, $T \leftarrow T.\text{fold}(\{T'[v_{i+(j-1)k}] : j \in [h/k]\})$
end for
for each child v of $\text{root}(T)$ **do**
    Replace $T[v] \leftarrow \text{FairHC}(T[v])$
end for
Return T'

To see why this creates a fair, low-cost hierarchy, we first bound the metrics on the operators used. When we execute level abstraction, we can leverage relative balance and Lemma 59 to show that during FairHC, we can bound the abstraction operation cost.

Lemma 64. *In Algorithm 8, the level abstraction has operation cost at most* $e^{2/c}h$.

Our tree folding operation cost bound also comes from the balance of a tree, since any two vertices that are folded together must be at similar depths.

Lemma 65. *In Algorithm 8, each tree folding has operation cost at most* $ke^{4/(c(1-o(1))}$ *and acts on* k *trees.*

In order to bound the cost, we need to first know how many times an edge will be separated. We notice that an edge that is separated by level abstraction can no longer be separated on a subsequent recursive step. Additionally, the number of tree fold operators is proportionally bounded by the recursive depth, as it only happens λ times each step.

Lemma 66. *In Algorithm 8, an edge e is separated by at most 1 level abstraction and $\lambda \log_2(n)/\log_2(h)$ tree folds. The maximum recursion depth is also at most $\log_2(n)/\log_2(h)$.*

Fairness comes from the ordering over ℓ-colored vertices and the way select subtrees to fold together. One recursive step of FairHC incurs a small constant factor proportionate loss in potential fairness, and the number of times this loss occurs is bounded by the depth of recursion. We desire these fractions to be close to the true color proportions, which we can get arbitrarily close to by setting parameters c, k, and h.

Lemma 67. *For an ϵ-relatively balanced hierarchy T over $\ell(V) = c_\ell n = O(n)$ vertices of each color $\ell \in [\lambda]$, Algorithm 8 yields a hierarchical clustering T' such that the amount of each color $\ell \in [\lambda]$ in each cluster (represented by vertex v) is bounded as follows:*

$$\frac{c_\ell}{e^{2\log_h n/c}} \leq \frac{\ell(v)}{n_T(v)} \leq c_\ell \cdot (e^{4/c}/(kc_\ell) + e^{6/c})^{\log_h n}.$$

In Theorem 63, fairness is a result of Lemma 67, the operator properties are a result of Lemmas 64, 65, and 66, and the approximation factor has already been worked out by Lemma 76.

3.5 Experiments

This section validates our algorithms from Section 3.4. Our simulations demonstrate that our algorithm incurs only a modest loss in the hierarchical clustering objective and exhibits increased fairness. Specifically, the approximate cost increases as a function of Algorithm 8's defining parameters: c, δ, and k.

Datasets. We use two data sets, *Census* and *Bank*, from the UCI data repository [Dua and Graff, 2017]. Within each, we subsample only the features with numerical values. To compute the *cost* of a hierarchical clustering we set the similarity to be $w(i,j) = \frac{1}{1+d(i,j)}$ where $d(i,j)$ is the Euclidean distance between points i and j. We color data based on binary (represented as blue and red) protected features: *race* for *Census* and *marital status* for *Bank* (both in line with the prior work of [Ahmadian et al., 2020b]). As a result, *Census* has a blue to red ratio of 1:7 while *Bank* has 1:3.

We then subsample each color in each data set such that we retain (approximately) the data's original balance. We use samples of size 256. For each experiment, we do 10 replications and report the average results. We vary the parameters $c \in \{2^i\}_{i=0}^{5}, \delta \in (\frac{1}{8}, \frac{7}{8})$, and $k \in \{2^i\}_{i=1}^{4}$ to experimentally validate their theoretical impact on the approximate guarantees of Section 3.4.

Implementation. The Python code for the following experiments are available in the Supplementary Material. We start by running average-linkage, a popular hierarchical clustering algorithm. We then apply Algorithms 6 - 8 to modify this structure and induce a *fair* hierarchical clustering that exhibits a mild increase in the cost objective.

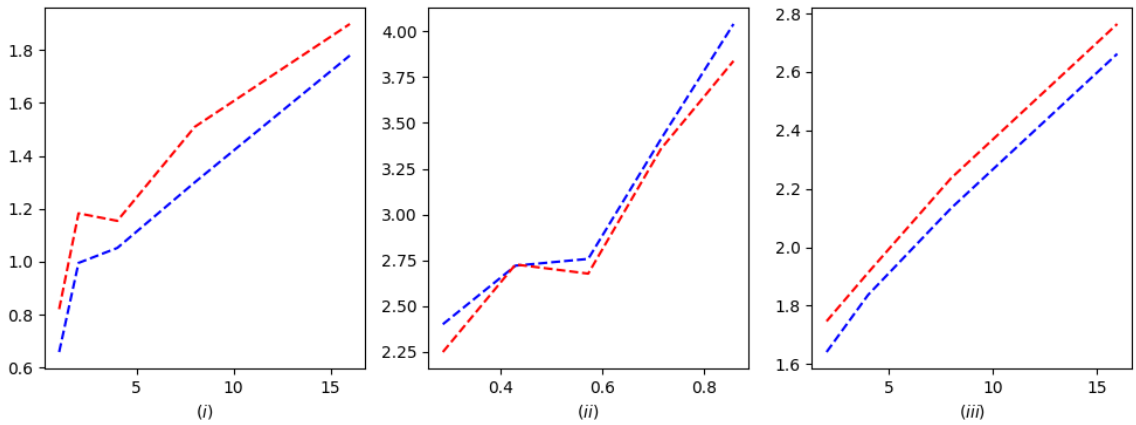


Figure 3.4: Cost ratio of Algorithm 8 as compared to average-linkage. (i) Ratio increase as a function of the parameter c, (ii) ratio increase as a function of the parameter δ, and (iii) ratio increase as a function of k. Blue lines indicate the result for *Census* dataset whereas red indicates the *Bank* dataset results.

Metrics. In our results we track the approximate cost objective increase as follows: Let G be our given graph, T be average-linkage's output, and T' be Algorithm 8's output. We then measure the ratio $\text{RATIO}_{cost} = \frac{cost_G(T')}{cost_G(T)}$.

Results. We first note that the average-linkage algorithm must construct unfair trees since, for each data set, the algorithm induces some monochromatic clusters. Thus, our resultant fair clustering is of considerable value in practice.

In Figure 1, we plot the change in cost ratio as the parameters (c, δ, k) are varied for the two datasets. Supporting our theoretical results, increasing our fairness parameters leads to a modest increase in cost. This is an empirical illustration of our fairness-cost approximation tradeoff according to our parameterization. Note that the results are consistent across tested datasets.

We additionally illustrate the resulting balance of our hierarchical clustering algorithm by presenting the distribution of the cluster ratios of the projected features (blue to red points) in Figure 3.5 for the *Census* data. The output of average-linkage naturally yields an unfair clustering of the data, yet after applying our algorithm on top this hierarchy we see that the cluster's balance

move to concentrate about the underlying data balance of 1:7. An equivalent figure for the *Bank* dataset is provided in the appendix due to space constraints.

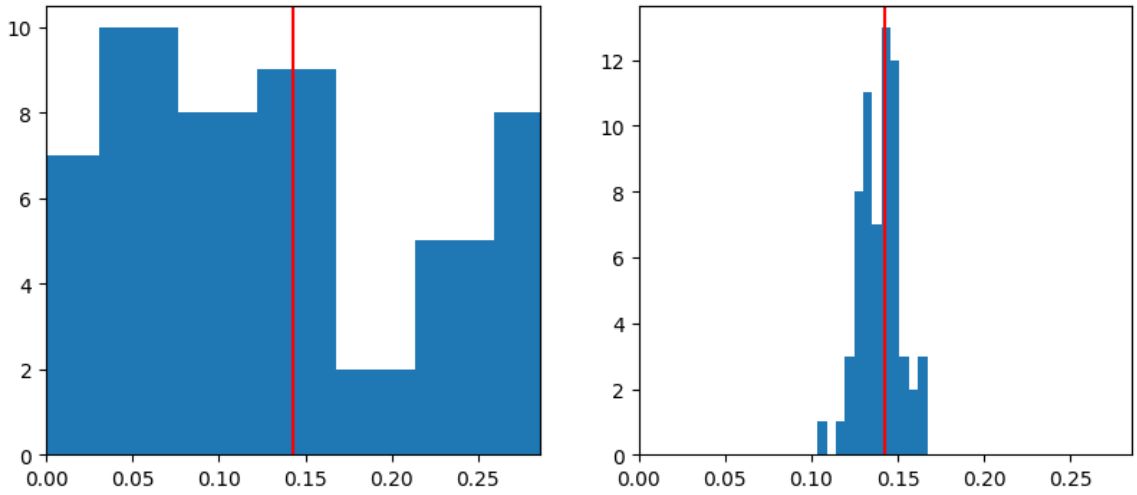


Figure 3.5: Histogram of cluster balances after tree manipulation by Algorithm 8. The left plot depicts the balances after applying the average- linkage algorithm and the right shows the result of applying our algorithm. The vertical red line indicates the balance of the dataset. Parameters were set to $c = 4, \delta = \frac{3}{8}, k = 4$ for the above clustering result.

3.6 Limitations

The main limitations this work suffers from encapsulate the general limitations of study in theoretical clustering fairness. Our work strives to provide algorithms that are applicable to many hierarchical clustering applications where fairness is a concern. However, our work is inherently limited by its focus on a specific fairness constraint (i.e., the extension of disparate impact originally used to study fair clustering [Chierichetti et al., 2017]). While disparate impact has received substantive attention in the clustering community and is seen as one of the primary fairness definitions/constraints [Ahmadian et al., 2020b, Ahmadian et al., 2020c, Bera et al., 2019, Brubach et al., 2020, Kleindessner et al., 2019b], it is just one of many established fairness constraints for problems in clustering [Chakrabarti et al., 2021, Chen et al., 2019, Esmaeili et al., 2021, Klein-

dessner et al., 2019a]. When applying fair machine learning algorithms to problems, it is not always clear which fairness constraints are the best for the application. This, and the fact that the application of fairness to a problem can cause harm in other ways [Ben-Porat et al., 2021], means that the proposal of theoretical fair machine learning algorithms always has the potential for improper or even harmful use. While this work proposes purely theoretical advances to the field, we direct the reader to [Barocas et al., 2019] for a broader view on the field.

Our results are also limited by the theoretical assumptions that we make. For instance, in the stochastic fairness algorithm, we assume that the probabilities of a vertex being a certain color are within the same bounds across all vertices. This may not be realistic, as there could be higher variance in the distribution of color probabilities, and even though the probabilities may lie outside of our assumed bounds, it still may be tractable to find a low-cost hierarchical clustering.

In our main theorem, we assume that there are only two colors (protected classes), and that they subsume a constant fraction of the general population. The former assumption is clearly limited in that in many cases, protected classes may take on more than two values. The constant fraction assumption is actually highly relevant and is reflected in other clustering literature, but it is a potential limitation that may rule out a handful of applications nevertheless.

Finally, our results are limited to the evaluation of hierarchical clustering quality based off cost. While this is a highly regarded metric for hierarchy evaluation, there may be situations where others are appropriate. It also neglects the practicality of empirical study in that many important machine learning algorithms we use today cannot provide guarantees across all data (which our results necessarily do), but they perform much better on most actual inputs. However, we leave it as an open question to further evaluate the practicality of our algorithms through empirical study.

3.7 Proofs: Tree Properties and Operators

Here we present all our proofs and theoretical results regarding our tree operator properties.

We start by discussing our tree rebalance operator. Effectively, any edge whose end points are separated by a tree rebalance operator was contained in a cluster of size $n_T(u)$, and now we guarantee they are in a cluster of size $n_T(v)$.

Lemma 68. *Given a tree T, let $T' = \text{tree\_rebalance}(u, v)$ for a node u and an ancestor node v. The only edges separated by this are $e = (x, y)$ such that $x \in \text{cluster}(u)$ and $y \in \text{cluster}(a) \setminus \text{cluster}(u)$. The operation cost is bounded above by $\Delta = n_T(v)/n_T(p)$, where p is the parent of u.*

Proof of Lemma 68. Let $e = (x, y)$ be an edge that is separated by a tree rebalance operator $\text{tree\_rebalance}(u, v)$ for some internal nodes u and v. Let's consider when we execute the rebalance. Let $V = \text{cluster}(v)$ be the set of vertices corresponding to V. Traverse down the tree from v to u. Label the clusters we come across $A_1, A_2, \ldots, A_{k-1}$ and their corresponding un-traversed children $B_1, B_2, \ldots, B_{k-1}$. Let $A_k = \text{cluster}(u)$, and B_k be the cluster for its only sibling.

When we rebalance, our first split will now divide V into A_k and $B := \cup_{i \in [k]} B_i$. For e to be separated by the rebalance of u with respect to v, it must be that $x \in \text{cluster}(u)$ and $y \in B = \text{cluster}(a) \setminus \text{cluster}(u)$ (without loss of generality). This means that their lowest common ancestor was on the path between u and v (excluding u), which means the smallest $n_T(e)$ could be is $n_T(p)$ where p is the parent of u. That means $\text{cost}_T(e) = w(e) \cdot n_T(p)$.

In T', their lowest common ancestor is v, thus $n_{T'}(e) = n_{T'}(v) = n_T(v)$, following from the observation that v's cluster does not change. Thus, $\text{cost}_{T'}(e) \leq w(e) \cdot n_T(v)$. Putting these

together, we find $\text{cost}_{T'}(e) \leq \frac{n_T(v)}{n_T(p)} \text{cost}_T(e)$. □

For our subtree deletion and insertion, the idea is that an edge that is separated costs at least $n_T(v)$ in the original tree, but may cost up to $n_T(u \wedge v)$ in the modified tree.

Lemma 69. *Given a tree T, let $T' = \text{del\_ins}(u, v)$ for two nodes u and v, where u is not an ancestor of v. The only edges separated by this are $e = (x, y)$ such that $x \in \text{cluster}(u)$ and $y \in \text{cluster}(u \wedge v) \setminus \text{cluster}(u)$. The operation cost is bounded above by $\Delta = n_T(u \wedge v)/n_T(u)$.*

Proof of Lemma 69. Let $e = (x, y)$ be an edge that is separated by a subtree deletion and insertion operators $\text{del\_ins}(u, v)$ for some appropriate internal nodes u and v. Let's consider when we execute the subtree deletion and insertion. For x and y to be separated, x must be in $\text{cluster}(u)$ and y must be in $\text{cluster}(u \wedge v) \setminus \text{cluster}(u)$ (without loss of generality). The first part is true because only the subtree $T[u]$ is moved, otherwise their least common ancestor would be unaffected. The second part is true because otherwise y is either in $T[u]$ too, in which case their relative position remains the same in the subtree, or $y \notin T[u \wedge v]$, in which case still the move still does not affect their least common ancestor (which is higher in the tree than $u \wedge v$).

Now, since $x \in T[u]$ and $y \notin T[u]$, $x \wedge y$ must be an ancestor of u, thus $n_T(e) \geq n_T(u)$. This means that $\text{cost}_T(e) \geq w(e) \cdot n_T(u)$. In T', their least common ancestor must still remain below $u \wedge v$, since all the points in $T[u \wedge v]$ remain somewhere below $u \wedge v$. Also note no points are added to $T[u \wedge v]$ over the two operators. Thus $n_{T'}(e) \leq n'_T(u \wedge v) = n_T(u \wedge v)$. This means $\text{cost}_{T'}(e) \leq w(e) \cdot n_T(u \wedge v)$, so $\text{cost}_{T'}(e) \leq \frac{n_T(u \wedge v)}{n_T(u)} \text{cost}_T(e)$. Thus, $\Delta = \frac{n_T(u \wedge v)}{n_T(u)}$. □

The level abstraction operator is somewhat more complicated, as it modifies entire levels of the tree, instead of individual splits. However, we can still use our notion of operation cost to

bound the operator's impact. This just becomes a bit more vague because we have to look at the largest and smallest clusters between depths h_1 and h_2 in T.

Lemma 70. *Say we apply the level abstraction operator between heights h_1 and h_2 on hierarchy T to yield T'. An edge is separated by the operator if and only if the least common ancestor of its endpoints is between h_1 and h_2. Its operation cost is at most $\Delta \leq \frac{n_T(u)}{n_T(v)}$, where u and v are two clusters that are abstracted away that maximize this ratio.*

Proof of Lemma 70. Let $e = (x, y)$ be an edge that is separated by a level abstraction operator $\text{abstract}(h_1, h_2)$ for some depths h_1 and h_2 with $h_1 < h_2$. Let's consider when we execute the abstraction. For x and y to be separated, $x \wedge y$ must be merged into its parent by the operator. That means it is between depth h_1 and h_2. Let v be the vertex with the smallest $n_T(v)$ between depths h_1 and h_2. Then $n_T(x \wedge y) \geq n_T(v)$, and so $\text{cost}_T(e) \geq w(e) \cdot n_T(v)$.

The ancestor it eventually gets contracted into must be of depth h_1, because we stop contracting after that point. Although its tree structure is altered below it, its cluster size remains the same since no vertices are moved away or to its subtree. Let u be the vertex with the largest $n_T(u)$ between depths h_1 and h_2. Then we get $n_{T'}(x \wedge y) \geq n_T(u)$, and so $\text{cost}_{T'}(e) \leq w(e) \cdot n_T(u)$.

This has shown us that $\text{cost}_{T'}(e) \leq \frac{n_T(u)}{n_T(v)} \text{cost}_T(e)$. Notice that u and v are precisely the internal nodes that maximize the ratio, so $\text{cost}_{T'}(e) \leq \frac{n_T(u)}{n_T(v)} \text{cost}_T(e)$.

□

Tree folding is a bit more complicated because we are merging multiple clusters on top of each other. Thus we have to factor in the value k on top of considering varying cluster sizes. Ultimately, however, the product of the ratio between cluster size and k bound the proportional increase in cost.

Lemma 71. *Say we apply the tree folding operator on hierarchy T to yield T'. Its operation cost is at most $\Delta \leq k\frac{n_T(u)}{n_T(v)}$, where u and v are two clusters that are mapped to each other away that maximize this ratio.*

Proof of Lemma 71. Let $e = (x, y)$ be an edge that is separated by a tree folding operator $\text{fold}(T_1, \ldots, T_k)$ for subtrees $T_1, \ldots, T_k$ of T satisfying the operator conditions. Let's consider when we execute the folding. For x and y to be separated, $x \wedge y$ must be in one of the subtrees, say T_1 without loss of generality. This means $\text{cost}_T(e) \geq w(e) \cdot n_T(x \wedge y)$.

Now we consider the cost in T'. Clearly, $x \wedge y$ becomes the single vertex in T_f corresponding to $\phi_1(x \wedge y)$. A leaf vertex in T_2 (without loss of generality) is only a descendant of $\phi_1(x \wedge y)$ if it has an ancestor a such that $\phi_2(a) = \phi_1(x \wedge y)$. Therefore:

$$n_{T'}(x \wedge y) = n_{T'}(\phi_1(x \wedge y)) \leq \sum_{i \in [k]} n_T(\phi_i^{-1}(\phi_1(x \wedge y))$$

If $u = \max\{n_T(u) : u \in T_i, i \in [k], \phi_i(u) = \phi_1(x \wedge y)\}$, then we further have:

$$n_{T'}(x \wedge y) \leq \sum_{i \in [k]} n_T(u) = k n_T(u)$$

This means that $\text{cost}_{T'}(e) \leq w(e) \cdot k n_T(u)$. Putting these together gives $\text{cost}_{T'}(e) \leq \frac{n_T(x \wedge y)}{n_T(v)} k \, \text{cost}_T(e) \leq \frac{n_T(u)}{n_T(v)} k \, \text{cost}_T(e)$ where u and v are the vertices merged together that maximize this ratio.

□

3.8 Proofs: Results

In this section, we prove all lemmas, theorems, and missing algorithmic discussion regarding our main results.

3.8.1 RebalanceTree

This section contains the proofs regarding RebalanceTree.

Proof of Lemma 51. By our definition of A_i and B_i for all $i \in [k]$, $|A_{k-1}| \geq 2n/3$, implying $|A_k| \geq \frac{1}{2}|A_{k-1}| \geq n/3$, and also that $|A_k| \leq n/3$ since $|B_k| \geq n/3$. Thus $n/3 \leq |A_k|, |B| \leq 2n/3$. Since we rearrange our first split to be this way, that means our first tree rebalance creates a first split that satisfies the relatively balanced condition. From here, we recurse on each side, guaranteeing that one split after another satisfies the condition. Thus, the entire tree is $\frac{1}{6}$-relatively balanced. □

Proof of Lemma 52. Consider an edge $e = (x, y)$ that is first rebalanced at some recursive step in Algorithm 6. By Lemma 68, x and y must now be separated at the current tree's root. Therefore, at any further level of recursion, only one of e's endpoints will be present, so it cannot be separated again. □

Proof of Lemma 53. The rebalance operator is applied to v (the node found) at r. Notice that v's parent p must be such that $n_T(p) \geq \frac{2}{3}n_T(r)$, otherwise the loop would have stopped earlier. Therefore, by Lemma 68, the operation cost is $n_T(r)/n_T(p) \leq 3/2$. □

Proof of Theorem 50. Let T^* be the optimal tree, let T_1 be our guaranteed γ-approximation on T, and let T' be our output. By Lemma 51, T' is $1/6$-relatively balanced. By Lemmas 52 and 53,

every edge is separated by at most one tree rebalance operator of length at most $3/2$. Because of this, $\text{cost}_{T'}(e) \leq (3/2)\,\text{cost}_{T_1}(e)$. Summing over all edges yields $\text{cost}(T) \leq (3/2)\,\text{cost}(T_1) \leq \gamma\,\text{cost}(T^*)$. □

3.8.2 RefineRebalanceTree

This section contains the proofs regarding RefineRebalanceTree as well as the algorithmic description of SubtreeSearch.

Algorithm 9 SubtreeSearch

Input A $\frac{1}{6}$-relatively balanced hierarchy tree T of size n, with smaller cluster always on the left and error parameter s.

Output Modified $\frac{1}{6}$-relatively balanced T by a subtree deletion and insertion of a subtree of size between $s/3$ and s

$v = \text{root}(T)$
while $|\,\text{leaves}(\text{left}_T(v))| > s$ **do**
    $v \leftarrow \text{right}_T(v)$
end while
$v \leftarrow \text{left}_T(v)$

$u \leftarrow \text{root}(T)$
while $|\,\text{leaves}(\text{right}_T(u))| \geq |\,\text{leaves}(v)|$ **do**
    $u \leftarrow \text{left}_T(u)$
end while
$T' \leftarrow T.\text{del\_ins}(u, v)$
Return T

As discussed in the body, at a given split, SubtreeSearch traverse the tree below the larger cluster further down in a similar manner until we find a sufficiently small cluster. This cluster must be smaller than the current balance error, ϵ. We simply do this by always traversing to the larger cluster as in Algorithm 9 until its smaller child is sufficiently small. We then remove that subtree, traverse back to the top of the tree, and try to reinsert the subtree by recursing down the right children.

This exhibits nice properties with respect to relative balance.

Lemma 72. SubtreeSearch *preserves* $\frac{1}{6}$*-relative balance.*

Proof of Lemma 72. Consider T, the tree at the beginning of the algorithm, and let v be the vertex whose subtree we end up moving. To start, we only consider the deletion, and then we will consider the reinsertion of v's subtree. The only vertices whose corresponding cluster sizes are altered (specifically, reduced) are v's ancestors. Note that they are all right children (i.e., the bigger sibling at the start) and they are reduced by size $n_T(v)$.

Let p be the parent of v. Since $v = \text{left}_T(p)$, we know $n_T(v) \leq \frac{1}{2} n_T(p)$. Since we remove that many vertices, $n_T(p)$ is at worst halved. Since p is a right child, say with sibling node q, $n_T(p) \geq n_T(q)$ at the start. Then at the end, $n_{T'}(p) \geq \frac{1}{2} n_{T'}(q)$. This implies that, in the end, the clusters are between 1/3 and 2/3 the size of their parent. Thus relative balance is held on this split. For ancestor nodes a of p in T, this argument holds since $n_T(a) > n_T(p)$ both before and after, and a is also a right child. Therefore, the entire tree is still $\frac{1}{6}$-relatively balanced after subtree deletion.

Now we consider the second half of the algorithm, where we reinsert $T[v]$. Let u be the vertex we select to insert at, p be its new parent, g be its old parent (now its grandparent), and $r = \text{right}_T(g)$ be its old sibling. Before insertion, we know that $n_T(r) \geq n_T(v)$ by the while loop condition. Since T is $\frac{1}{6}$-relatively balanced still, and r is u's sibling, $n_T(u) \geq \frac{1}{2} n_T(r)$. Since the algorithm did not stop at p, then $n_T(r) \geq n_T(v)$, thus implying $n_T(u) \geq \frac{1}{2} n_T(v)$. Additionally, since the algorithm stopped on u, $n_T(\text{right}_T(u)) \leq n_T(v)$. Since that is u's larger child, $n_T(u) \leq 2n_T(\text{right}_T(u)) \leq 2n_T(v)$. Since v is u's new sibling, and one is not more than twice the size of the other, we have $\frac{1}{6}$-relative balance at that split.

The only other vertices impacted by the insertion are u's ancestors. For an ancestor node a of u in T, the argument also holds since $n_T(a) \geq n_T(p) \geq n_T(v)$ meaning $n_{T'}(a) \leq 2n_T(a)$ and a must be a left (and therefore smaller) child. Therefore, the relative balance is kept at all splits involving ancestors of u, thus we have relative balance. □

Our other guarantee is that we find a subtree of size at least $s/2$ to move. This comes from our first loop's end condition.

Lemma 73. *In Algorithm 9, $s/3 \leq n_T(v) \leq s$.*

Proof of Lemma 73. When the first loop stops, this is the first visited vertex whose left child, which ends up being the final v, is at most s. Thus $n_T(v) \leq s$. Since this was the first such instance, if g is the grandparent of v, this means $n_T(\text{left}_T(g)) > s$ since the loop continued after g. Since right children are larger and v's parent p is $\text{right}_T(g)$, $n_T(p) \geq n_T(\text{left}_T(g)) > s$. Since we have $\frac{1}{6}$-relative balance, $n_T(v) \geq \frac{1}{3} n_T(p) \geq \frac{1}{3} s$. □

Proof of Lemma 55. At each iteration of Algorithm 7, as long as the relative balance is above ϵ, we move a subtree of size at least $\frac{1}{3}\delta n$ and at most δn by Lemma 73 where δ is the current relative balance. This means that the relative balance of the first split reduces by a factor of $\frac{2}{3}$, and by Lemma 72, the rest of the tree remains $\frac{1}{6}$-relatively balanced. This is simply done until the relative balance of the first split is small enough. When we recurse, we are still guaranteed $\frac{1}{6}$-relatively balance, and we can then ensure all sufficiently large splits are ϵ-relatively balanced. □

Proof of Lemma 56. Consider an edge e that is first separated by some subtree deletion and insertion operator at some recursive step in Algorithm 7. Notice e must now be separated at the current tree's root. This means that at any further level of recursion, only one of e's end points will be present, so it cannot be separated again. □

Proof of Lemma 57. The subtree deletion and insertion operator is applied at u of $T[v]$ when $u \wedge v$ is the root, i.e., $n_T(u \wedge v) = n_T(r) \leq n$ where r is the current tree's root and n is our original data set size. We never allow the algorithm to continue with $\delta \leq \epsilon$, therefore the smallest tree size $T[v]$ that we move is $\frac{1}{3}n\epsilon$ by Lemma 73. Thus the operation cost is at most $n_T(u \wedge v)/n_T(v) \leq \frac{3}{\epsilon}$ by Lemma 69. □

Proof of Theorem 54. Let T^* be the optimal tree, let T_1 be our $1/6$-relatively balanced $3\gamma/2$ approximation guaranteed by Theorem 50, and let T' be our output. By Lemma 55, T' is ϵ-relatively balanced. By Lemmas 56 and 57, every edge is separated by at most one subtree deletion and insertion operator of operation cost at most $3/\epsilon$. Because of this and because of Lemma 69, $\text{cost}_{T'}(e) \leq \frac{3}{\epsilon}\,\text{cost}_{T_1}(e)$. Summing over all edges yields $\text{cost}(T) \leq \frac{3}{\epsilon}\,\text{cost}(T_1) \leq \frac{9\gamma}{2\epsilon}\,\text{cost}(T^*)$. □

3.8.3 StochasticallyFairHC

This section contains the proofs regarding StochasticallyFairHC.

Proof of Lemma 59. Since T is ϵ-relatively balanced, any cluster A that splits into clusters B and C satisfies $(1/2 - \epsilon)|A| \leq |C| \leq |B| \leq (1/2 + \epsilon)|A|$, without loss of generality. This means that the maximum cluster size that can be found at level i is bounded above by traversing the tree from root down assuming that we always traverse to a maximally sized child, e.g., if p is a parent of w on our path, then $n_T(w) \leq (1/2 + \epsilon)n_T(p)$.

Since we traverse i levels, we get for any i-level vertex u, $n_T(u) \leq (1/2 + \epsilon)^i n$. By the reverse logic (i.e., traversing from the root to a minimally sized child), for any i-level vertex v, $n_T(v) \geq (1/2 - \epsilon)^i n$. Then their ratio must be at most $\frac{n_T(u)}{n_T(v)} \leq \frac{(1+2\epsilon)^i}{(1-2\epsilon)^i}$.

Finally, consider if $i \leq \log_{1/2-\epsilon}(x/n)$. We can just assume it is at the maximum possible level, because this will clearly give the loosest bounds. We already know the smallest cluster size at level i is at least $(1/2-\epsilon)^i n$, and the ratio between the largest and smallest cluster sizes is at most $\left(\frac{1+2\epsilon}{1-2\epsilon}\right)^i$. Therefore, for a vertex u at level i:

$$n_T(u) \leq \left(\frac{1+2\epsilon}{1-2\epsilon}\right)^{\log_{1/2-\epsilon}(n/x)} (1/2-\epsilon)^{\log_{1/2-\epsilon}(x/n)} n$$

The second term in the product obviously simplifies to x. For the first term, we can see that since $\epsilon = 1/(c\log_2 n)$:

$$\frac{1+2\epsilon}{1-2\epsilon} = 1 + \frac{4\epsilon}{1-2\epsilon} = 1 + \frac{4}{c(1-2\epsilon)\log_2 n}$$

We can also bound the exponent. Note that we raise a value that is at least 1 to the exponent, so to create an upper bound, we must upper bound the exponent as well. We leverage the fact that $1/(1/2-\epsilon) > 2$ because $\epsilon \in (0, 1/2)$. This implies $\log_2(1/(1/2-\epsilon)) > 1$.

$$\begin{aligned}
\log_{1/2-\epsilon}(x/n) =& \frac{\log_2(x/n)}{\log_2(1/2-\epsilon)} \\
=& \frac{\log_2(n/x)}{\log_2(1/(1/2-\epsilon))} \\
\leq& \log_2(n/x) \\
\leq& \log_2(n)
\end{aligned}$$

Where the last step comes from the fact taht $x \geq 1$. We can now put this all together.

$$n_T(u) \leq \left(1 + \frac{4}{c(1-2\epsilon)\log_2 n}\right)^{\log_2(n)} x \leq x \cdot e^{4/(c(1-o(1))}$$

□

Proof of Lemma 60. By Lemma 59, the smallest cluster at depth $i \geq t$ is at most $(1/2+\epsilon)^t n$. If we assume a trivial cluster size lower bound of 1, this implies for any contracted internal nodes u and v in the level abstraction, $n_T(u)/n_T(v) \leq (1/2+\epsilon)^t n$. □

Proof of Lemma 61. By Lemma 59, the largest cluster at depth $i \leq t$ in T is at least $(1/2 - \epsilon)^t n$. When we execute level abstraction, this cluster size is not changed, but we know all other potentially smaller clusters are contracted into their parents. Thus, this is the smallest cluster size in T'. □

Proof of Lemma 62. Consider a vertex v at height 1. By Lemma 61, $n_{T'}(v) \geq (1/2-\epsilon)^t n = \frac{3(1-\delta)}{a\delta^2}\ln(cn)$. Fix some $\ell \in [\lambda]$. Let $X_{\ell v}$ count the number of vertices of color ℓ in $\text{leaves}(v)$. Note this is a sum of Bernoullis, so $\mathbb{E}[|X_{\ell v}|] = \sum_{u \in \text{cluster}(v)} p_\ell(u)$. Note that we are given that $p_\ell(u) \geq \frac{1}{1-\delta}\alpha_\ell$ for all u. This gives us the following bounds from Lemma 61:

$$\begin{aligned}
\mathbb{E}[X_{\ell v}] &\geq \frac{3(1-\delta)}{a\delta^2}\ln(cn) \cdot \frac{1}{1-\delta} a = \frac{3}{\delta^2}\ln(cn) \\
\mathbb{E}[X_{\ell v}] &\geq \frac{1}{1-\delta}\alpha_\ell n_{T'}(v) \\
\mathbb{E}[X_{\ell v}] &\leq \frac{1}{1+\delta}\beta_\ell n_{T'}(v)
\end{aligned}$$

Then by a Chernoff bound with δ as the error parameter:

$$
\begin{aligned}
&P(|X_{\ell v} - \mathbb{E}[X_{\ell v}]| \geq \delta \mathbb{E}[X_{\ell v}]) \\
&\qquad \leq 2\exp(-\mathbb{E}[X_{\ell v}]\delta^2/3) \\
&\qquad = 2\exp(-\frac{3}{\delta^2}\ln(cn)\delta^2/3) \\
&\qquad = \frac{2}{cn}
\end{aligned}
$$

Thus with probability at least $1 - \frac{2}{cn}$:

$$
\begin{aligned}
X_{\ell v} - \mathbb{E}[X_{\ell v}] &\leq \delta \mathbb{E}[X_{\ell v}] \\
X_{\ell v} &\leq (1+\delta)\mathbb{E}[X_{\ell v}] \\
&\leq (1+\delta) \cdot \frac{1}{1+\delta}\beta_\ell n_{T'}(v) \\
&\leq \beta_\ell n_{T'}(v)
\end{aligned}
$$

In other words, the cluster $\mathrm{leaves}(v)$ satisfies the upper bound for color ℓ. We also find that:

$$
\begin{aligned}
-X_{\ell v} + \mathbb{E}[X_{\ell v}] &\geq \delta \mathbb{E}[X_{\ell v}] \\
X_{\ell v} &\geq (1-\delta)\mathbb{E}[X_{\ell v}] \\
&\geq (1-\delta) \cdot \frac{1}{1-\delta}\alpha_\ell n_{T'}(v) \\
&\geq \alpha_\ell n_{T'}(v)
\end{aligned}
$$

Which means it also satisfies the upper bound. Let y be the number of internal nodes with leaf-children. Since we already saw the minimum such cluster size is $O(\log n)$ (since $a, \delta = O(1)$), then $y = O(n/\log n)$. Notice, also, that the vertices counted by y are the only ones we need to prove are fair, since taking the union of two fair clusters is fair. Thus, to show this is true for all ℓ and v, we take a union bound over all λ values of ℓ and y values of v. We then find that with probability at least $1 - \frac{2\lambda n/\log n}{cn} = 1 - \frac{2}{\log n}$, all height 1 clusters must be fair, meaning the entire hierarchy must be fair by the union-bound property.

□

Proof of Theorem 58. Let T^* be the optimal tree, let T_1 be our ϵ-relatively balanced $\frac{9\gamma}{2\epsilon}$ approximation guaranteed by Theorem 54, and let T' be our output. By Lemma 62, T' satisfies our fairness constraints. By Lemmas 60 and the fact that we only apply one operator, every edge is separated by at most one level abstraction operator of operation cost at most $(1/2 + \epsilon)^t n$, but we know from Lemma 59 that this is bounded above by $e^{4/(c(1-o(1))} \cdot \frac{3(1-\delta)\ln(cn)}{a\delta^2}$. Because of this, $\text{cost}_{T'}(e) \leq e^{4/(c(1-o(1))} \cdot \frac{3(1-\delta)\ln(cn)}{a\delta^2} \text{cost}_{T_1}(e)$. Summing over all edges yields $\text{cost}(T) \leq e^{4/(c(1-o(1))} \cdot \frac{3(1-\delta)\ln(cn)}{a\delta^2} \text{cost}(T_1) \leq e^{4/(c(1-o(1))} \cdot \frac{3(1-\delta)\ln(cn)}{a\delta^2} \cdot \frac{9\gamma}{2\epsilon} \text{cost}(T^*)$. □

3.8.4 FairHC

This section contains the proofs and additional theoretical discussion regarding FairHC.

Lemma 74. StochasticallyFairHC *with* $t = \log_{1/2-\epsilon}(1/(2n\epsilon))$ *outputs a hierarchy where the* ϵ*-relatively balanced guarantee holds for all splits except those forming the leaves. Additionally, it admits a proportional cost increase of at most* $\frac{1}{2}ce^{4/(c(1-o(1))} \log_2 n$.

Proof of Lemma 74. Say T is our input (i.e., it is ϵ-relatively balanced). Notice that StochasticallyFairHC

only modifies T's structure below depth $\log_{1/2-\epsilon}(1/(2n\epsilon))$, which means any balance guarantees hold up to that level. By Lemma 59, for any vertex v at depth $\log_{1/2-\epsilon}(1/(2n\epsilon))$ or above, $n_T(v) \geq (1/2-\epsilon)^{\log_{1/2-\epsilon}(1/(2n\epsilon))}n = 1/(2\epsilon)$. By the definition of ϵ-relative balance, this means that the balance guarantee holds for the split at this vertex. Since all internal vertices in the resulting tree T' are at or above this level, all internal vertices except those with leaf children exhibit the relative balance guarantee.

In order to bound the proportional increase in cost, we must bound the operation cost of the level abstraction. The minimum depth in the abstraction is $\log_{1/2-\epsilon}(1/(2n\epsilon))$. By Lemma 59, this means the maximum cluster size is at most $e^{4/(c(1-o(1)))}/(2\epsilon) = \frac{1}{2}ce^{4/(c(1-o(1))}\log_2 n$. Since the smallest cluster size involved is at least 1, we can then bound the operation cost by this max cluster size, giving our result. □

Lemma 75. *If T is ϵ-relatively balanced besides the final layer of splits, then the subtrees rooted at all of the root's children in* FairHC *after tree folding are as well.*

Proof of Lemma 75. Tree folding only involves overlaying the topology of isomorphic trees (ignoring their leaves). Consider a non-root vertex v in FairHC after tree folding that is also not a parent of leaves. It is the result of merging k vertices $v_1, \ldots, v_k$, and its left and right children l and r are the result of merging $l_1, \ldots, l_k$ and $r_1, \ldots, r_k$ respectively. Due to this:

$$n_{T'}(v) = \sum_{i \in [k]} n_T(v_i)$$

$$n_{T'}(l) = \sum_{i \in [k]} n_T(l_i)$$

$$n_{T'}(r) = \sum_{i \in [k]} n_T(r_i)$$

We also have, by relative balance, for any $i \in [k]$:

$$(1/2 - \epsilon) n_T(v_i) \leq n_T(l_i), n_T(r_i) \leq (1/2 + \epsilon) n_T(v_i)$$

A simple combination of these shows that:

$$(1/2 - \epsilon) n_T(v) \leq n_T(l), n_T(r) \leq (1/2 + \epsilon) n_T(v)$$

This means the split from v to l and r is relatively balanced. We can apply this to all such splits to find the entire new subtree is relatively balanced. □

Proof of Lemma 64. By Lemma 59, for any vertex v at depth $i \geq \log_2 h$, $n_T(v) \geq (1/2 - \epsilon)^{\log_2 h} n$. This can be further simplified using that $\epsilon = 1/(c \log n)$ and $h \leq n$.

$$n_T(v) \geq \frac{1}{2^{\log_2 h}} \left(1 - \frac{2}{c \log_2 n}\right)^{\log_2 h} n \geq e^{-2/c} n/h$$

Obviously, the largest $n_T(u)$ for any u within our depth bounds is n. Thus the level abstraction operation cost is at most $n/(e^{-2/c} n/h) = e^{2/c} h$. □

Proof of Lemma 65. That it acts on k trees is obvious. To prove the operation cost, consider u, v in trees T_i and T_j respectively where $\phi_i(u) = \phi_j(v)$. Since we are using the tree isomorphism between the trees, this means that u and v have the same height in T_i and T_j respectively, which also means that they had the same height in the original tree T, as the roots of T_i and T_j are both at height $\log_2(h)$ in T. Since u and v are on the same level $i \leq \log_{1/2-\epsilon}(1/(2n\epsilon))$, Lemma 59 tells us:

$$\frac{n_T(u)}{n_T(v)} \leq e^{4/(c(1-o(1))}$$

Since this holds for all such pairs u and v, this also bounds the tree folding operation cost. Note that when we do this operator, the ϵ-relative balance is held by Lemma 75. Thus, this argument holds across all tree folds in the for loop. □

Proof of Lemma 66. If an edge $e = (u, v)$ is separated by the level abstraction, that means $u \wedge v$ is above depth $\log_2 h$. Notice that we recurse on clusters at depth $\log_2 h$, which means on any recursive instance here forward, e will not be contained within the trees, so e cannot be separated. Therefore, e can only be separated by one level abstraction.

Otherwise, notice that the depth of the last internal node is $\log_{1/2-\epsilon}(1/(2n\epsilon))$ by assumption. At each recursive step, we reduce the depth by $\log_2 h$ since we start at subtrees of depth h in the previous tree. Therefore, there are at most $\log_{1/2-\epsilon}(1/(2n\epsilon))/\log_2 h$ levels of recursion. To simplify this, we use similar methods to Lemma 65. which allows us to bound the recursive depth by $\log_2(n)/\log_2(h)$.

At each level of recursion, since e is contained in only one tree, it is only separated by λ tree folds. This means e may only be separated by $\lambda \log_2(n)/\log_2(h)$ tree folds. □

Lemma 76. *For an ϵ-relatively balanced hierarchy T, Algorithm 8 outputs a tree T' such that:*

$$\text{cost}(T') \leq e^{\frac{4\lambda \log_2(n)}{c(1-o(1))\log_2 h} + \frac{2}{c}} \cdot h \,\text{cost}(T)$$

Proof of Lemma 76. Lemmas 64, 65, and 66 tell us an edge e must only be involved in at most 1 level abstraction of operation cost at most $e^{2/c}h$ and $\lambda \log_2(n)/\log_2(h)$ tree folds of operation cost at most $e^{4/(c(1-o(1))}$ on k trees. By Lemmas 70 and 71, this will incur a total proportional cost increase of:

$$\frac{\text{cost}_{T'}(e)}{\text{cost}_T(e)} \leq (e^{4/(c(1-o(1))})^{\lambda \log_2(n)/\log_2(h)} \cdot e^{2/c}h$$

Which, summed over all edges, is equivalent to the desired result.

□

Lemma 77. *For an ϵ-relatively balanced hierarchy T over $\ell(V) = c_\ell n = O(n)$ vertices of each color $\ell \in [\lambda]$,* FairHC *before recursion ensures that the clustering induced on each depth-1 internal node v of the output tree T' each have $\frac{c_\ell}{e^{6/c}} \leq \frac{\ell(v)}{\text{leaves}(v)} \leq c_\ell \cdot (e^{4/c}/(kc_\ell) + e^{6/c})$ for each $\ell \in [\lambda]$.*

Proof of Lemma 77. We start by looking at one tree fold operator. Assume the color we are trying to sort is red. Consider the ordering of the vertices $\{v_i\}_{i\in[h]}$ from Algorithm 8, and let r_i the number of red points from $\text{leaves}(v_i)$ and R be the total number of red vertices.

Fix some i and let v_i' be the root vertex of the resulting subtree in the ith fold (i.e., the one all the subtrees are mapped onto). We know the vertices involved in this were $v_{i+(j-1)k}$ for all $j \in [h/k]$. Because of the ordering, we know that:

$$r_{i+(j-1)k}/n_T(v_{i+(j-1)k}) \leq r_{y+(j-2)k}/n_T(v_{y+(j-2)k}), \tag{1}$$

$$r_{i+(j-1)k}/n_T(v_{i+(j-1)k}) \geq r_{y+jk}/n_T(v_{y+(j-2)k}) \tag{2}$$

for all $y \in [h/k]$ assuming $j > 1$ for (1) and $j < k$ for (2). Since these three vertices are at the same height, say h' (with respect to T after rebalancing and before the algorithm began), Lemma 59 gives us that:

$$n_T(v_{i+(j-1)k})/n_T(v_{y+(j-2)k}) \leq \frac{(1+2\epsilon)^{\log_2 n}}{(1-2\epsilon)^{\log_2 n}},$$
$$n_T(v_{i+(j-1)k})/n_T(v_{y+jk}) \geq \frac{(1-2\epsilon)^{\log_2 n}}{(1+2\epsilon)^{\log_2 n}}$$

Combining these with the previous inequalities yield:

$$r_{i+(j-1)k} \leq \frac{(1+2\epsilon)^{\log_2 n}}{(1-2\epsilon)^{\log_2 n}} r_{y+(j-2)k},$$
$$r_{i+(j-1)k} \geq \frac{(1-2\epsilon)^{\log_2 n}}{(1+2\epsilon)^{\log_2 n}} r_{y+jk}$$

for all $y \in [h/k]$. Since $\epsilon = 1/(c \log_2 n)$, this bound can be further simplified to:

$$e^{-4/c} r_{y+jk} \leq r_{i+(j-1)k} \leq e^{4/c} r_{y+(j-2)k}$$

Since these hold for all y, we can say that:

$$\frac{k}{h}e^{-4/c}\sum_{y\in[h/k]} r_{y+jk} \leq r_{i+(j-1)k} \leq \frac{k}{h}e^{4/c}\sum_{y\in[h/k]} r_{y+(j-2)k}$$

Another way to think of this is partitioning the vertices (in order) into contiguous chunks of size h/k. Then $v_{i+(j-1)k}$ is the ith vertex in the jth chunk, and we know it has a lower of fraction of red points than clusters in the previous ($(j-2)$th) chunk and a higher fraction than clusters in the next (jth) chunk.

Now let R_{j-1} be the number of reds in the entire jth chunk (i.e., $R_{j-1} = \sum_{y\in[h/k]} r_{y+(j-1)k}$) Additionally, we can make a comparison between the reds in all chunks and R, namely, $\sum_{j\in[k]} R_{j-1} = R$.

Putting our two previous results together, for our fixed i:

$$\begin{aligned}
\sum_{j\in[k]} r_{i+(j-1)k} \leq & r_i + \frac{k}{h}e^{4/c}\sum_{j\in[k]} R_{j-1} \\
= & r_i + \frac{k}{h}e^{4/c}R, \\
\sum_{j\in[k]} r_{i+(j-1)k} \geq & r_{h-h/k+i} + \frac{k}{h}e^{-4/c}\sum_{j\in[k]} R_{j-1} \\
= & \frac{k}{h}e^{-4/c}R
\end{aligned}$$

Notice that if everything were perfectly balanced, $\frac{k}{h}R$ is exactly the number of reds we would want in $\text{leaves}(v_i')$. We now must bound r_i. Unfortunately, it could be an entirely red

cluster, so this is only bounded by the size of the cluster at depth $\log_2 h$, which we get from Lemma 59.

$$\begin{aligned} r_i \leq & n_T(v_i) \\ \leq & (1/2+\epsilon)^{\log_2 h} n \\ = & 2^{-\log_2 h}(1+2\epsilon)^{\log_2 h} n \\ \leq & e^{2/c} n/h \end{aligned}$$

Note the final inequality comes from the fact that $h \leq n$ and $\epsilon = 1/(c \log n)$. Now note that we are given $R = c_R n$ for some $c_R = O(1)$. We can sub this in.

$$r_i \leq e^{2/c} R/(c_R h)$$

Now, notice we are actually looking for the fraction of red points in the cluster. Since Lemma 59 gives us that $n_{T'}(v_i') \geq k(1-2\epsilon)^{\log_2 h} n/h \geq k e^{-2/c} n/h$ and $n_{T'}(v_i') \leq k(1+2\epsilon)^{\log_2 h} n/h \leq k e^{2/c} n/h$ (applying the same logic as the upper bound to $n_T(v_i)$, k times), we get:

$$\begin{aligned}\frac{\sum_{j\in[k]} r_{i+jk}}{n_{T'}(v_i')} \leq& \frac{e^{2/c}R/(c_R h) + \frac{k}{h}e^{4/c}R}{ke^{-2/c}(n/h)}\\ =& \frac{R}{n}\cdot\left(\frac{e^{4/c}/c_R}{k} + e^{6/c}\right),\\ \frac{\sum_{j\in[k]} r_{i+jk}}{n_{T'}(v_i')} \geq& \frac{\frac{k}{h}e^{-4/c}R}{ke^{2/c}(n/h)}\\ =& \frac{R}{n}\cdot\frac{1}{e^{6/c}}\end{aligned}$$

This completes the proof for one tree fold under the observation that $\frac{R}{n} = c_\ell$ if red is ℓ. The same (if not stronger) bounds hold for all subsequent λ tree folds for each color. Note that as we proceed, this bound will not be disrupted since merging two clusters that guarantees the same upper bound on the fraction of red points still guarantees the same bound. □

Proof of Lemma 67. Clearly, the most imbalanced clusters in this process will be the clusters in the final level of the hierarchy. By Lemma 77, when we recurse, we have at most an $\frac{c_\ell}{e^{6/c}} \leq \frac{\ell(v)}{\text{leaves}(v)} \leq c_\ell \cdot (e^{4/c}/(kc_\ell) + e^{6/c})$ fraction of vertices of color ℓ for each $\ell \in [\lambda]$. Clearly, after at most $\log_2 n/\log_2 h = \log_h n$ recursive levels guaranteed by Lemma 66, our bound becomes:

$$\frac{c_\ell}{e^{6t\log_h n/c}} \leq \frac{\ell(v)}{\text{leaves}(v)} \leq c_\ell \cdot (e^{4/c}/(kc_\ell) + e^{6/c})^{\log_h n}.$$

□

Proof of Theorem 63. Let T^* be the optimal tree, let T_1 be our input tree which is a $ce^{4/(c(1-o(1))}\log_2 n \cdot \frac{9\gamma}{4\epsilon}$ approximation guaranteed by Theorem 58 but using $t = \log_{1/2-\epsilon}(1/(2n\epsilon))$ (this was shown more explicitly in Lemma 74), and let T' be our output. By Lemma 67, T' satisfies our fairness

constraints. By Lemmas 64, 65, and 66, every edge is separated by at most 1 level abstraction of max operation cost $e^{2/c}n/h$ and $\log_2(n)/\log_2(h)$ tree folds of operation cost at most $e^{4/(c(1-o(1))}$ on k subtrees. Lemma 76 immediately tells us:

$$\text{cost}(T') \leq e^{\frac{4\lambda \log_2 n}{c(1-o(1))\log_2 h}+\frac{2}{c}} \cdot h \,\text{cost}(T_1)$$

Combining this with the approximation guaranteed by T_1:

$$\begin{aligned}&\text{cost}(T')\\ &\leq e^{\frac{4\lambda \log_2 n}{c(1-o(1))\log_2 h}+\frac{2}{c}} \cdot hce^{4/(c(1-o(1))} \log_2 n \cdot \frac{9\gamma}{4\epsilon}\,\text{cost}(T^*)\end{aligned}$$

Simplifying and plugging in $h = n^\delta, \epsilon = 1/(c\log_2 n)$ yields the desired result. □

3.9 Runtime

Here we analyze the runtime of our four algorithms. Recall that before each of these algorithms, we run a black-box cost-approximate hierarchical clustering algorithm as well as all previous algorithms. For simplicity, here we will present the contribution of each algorithm to the runtime.

Theorem 50: This algorithm starts at the root, traverses down one side of the tree until a certain sized cluster is found, and then applies a tree rebalance. It then recurses on each child. The length of traversal is bounded by $O(n)$, and a single tree rebalance operation requires some simple constant-time pointer operations. As this is run from each vertex in the tree, the total

runtime is $O(n^2)$.

Theorem 54: As in the previous algorithm, here we do a computation at each of the $O(n)$ nodes in the tree. At each node, we apply subtree search until the desired balance is achieved. If δ is the current balance, we reduce this to at most $2\delta/3$ at each step. Thus, this will require a total of $O(\log(1/\epsilon))$ steps to complete. Each subtree search operation requires two, $O(n)$-length traversals to find the place to insert and delete. Otherwise, it is constant-time pointer math. Thus, the algorithm runs in $O(n^2 \log(1/\epsilon))$ time.

Theorem 58: This algorithm is quite, simple, as we are simply deleting some set of low nodes in the tree. Thus it only requires $O(n)$ time.

Theorem 63: Again, we execute a computation for at most $O(n)$ nodes in the tree. By a similar logic as before, tree abstraction steps require $O(n)$ time. It is not too hard to see that computing the fraction of red and blue vertices in each considered cluster and then sorting them accordingly also requires $O(n)$ time. Finally, we fold the vertices on top of each other. The isomorphism used for folding can be found by simply indexing the vertices in each subtree, and then applied quite directly, which also takes $O(n)$ time. Thus this requires only $O(n^2)$ time.

Therefore, the entire final algorithm (without the blackbox step) is bounded by the computation time from Theorem 54, which is $O(n^2 \log(1/\epsilon))$. Intuitively, ϵ is bounded by $\epsilon > 1/n$, thus this becomes $O(n^2 \log n)$.

Chapter 4: Fair, Polylog-Approximate Low-Cost Hierarchical Clustering

4.1 Introduction

Clustering is a pervasive machine learning technique which has filled a vital niche in every day computer systems. Extending upon this, a *hierarchical clustering* is a recursively defined clustering where each cluster is partitioned into two or more clusters, and so on. This adds structure to flat clustering, giving an algorithm the ability to depict data similarity at different resolutions as well as an ancestral relationship between data points, as in the phylogenetic tree [Kraskov et al., 2003].

On top of computational biology, hierarchical clustering has found various uses across computer imaging [Selvan et al., 2005], computer security [Chen et al., 2020b,Chen et al., 2021], natural language processing [Ramanath et al., 2013], and much more. Moreover, it can be applied to any flat clustering problem where the number of desired clusters is not given. Specifically, a hierarchical clustering can be viewed as a structure of clusterings at different resolutions that all agree with each other (i.e., two points clustered together in a higher resolution clustering will also be together in a lower resolution clustering). Generally, hierarchical clustering techniques are quite impactful on modern technology, and it is important to guarantee they are both effective and unharmful.

Researchers have recognized the harmful biases unchecked machine learning programs

pose. A few examples depicting racial discrimination include allocation of health care [Ledford, 2019], presentation of ads suggestive of arrest records [Sweeney, 2013], prediction of hiring success [Bogen and Rieke, 2018], and estimation of recidivism risk [Angwin et al., 2016]. A popular solution that has been extensively studied in the past decade is *fair machine learning*. Here, fairness concerns the mitigation of bias, particularly against protected classes. Most often, fairness is an additional constraint on the allowed solution space; we optimize for problems in light of this constraint. For instance, the notion of *individual fairness* introduced by the foundational work of [Dwork et al., 2012] deems that an output must guarantee that any two individuals who are similar are classified similarly.

In line with previous work in clustering and hierarchical clustering, this paper utilizes the notion of *group fairness*, which enforces that different protected classes receive a proportionally similar distribution of classifications (in our case, cluster placement). [Chierichetti et al., 2017] first introduced this as a constraint for the flat clustering problem, arguing that it mitigates a system's disparate impact, or non-proportional impact on different demographics. This notion of fair clustering has been similarly leveraged and extended by a vast range of works in both flat [Ahmadian et al., 2019, Bera et al., 2019, Bercea et al., 2019] and hierarchical [Ahmadian et al., 2020b, Knittel et al., 2023b] clustering.

To illustrate our fairness concept, consider the following application (Figure 4.1): a news database is structured as a hierarchical clustering of search terms, where a search term is associated with a cluster of news articles to output to the reader, and more specific search terms access finer-resolution clusters. When a user searches for a term, we simply identify the corresponding cluster and output the contained articles. However, as is, the system does not account for the political skew of articles. In Figure 4.1, we label conservative-leaning articles in red and

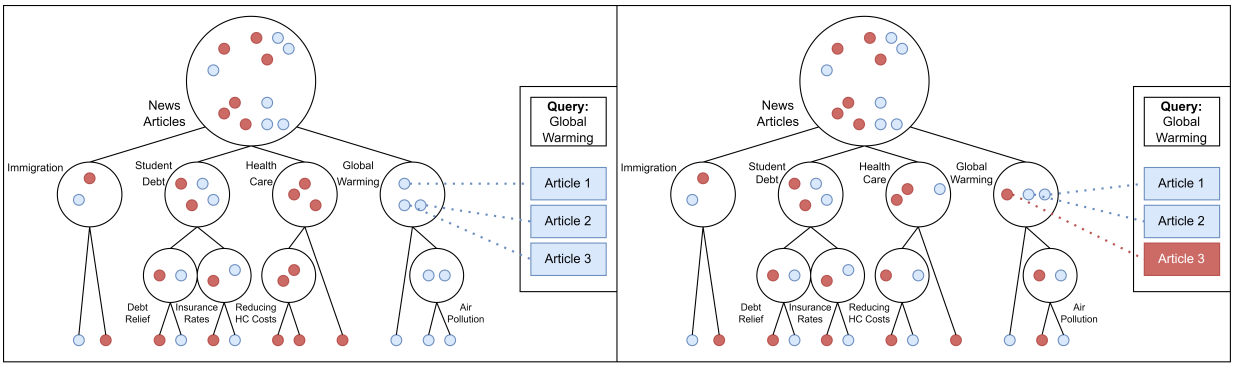


Figure 4.1: A hierarchical clustering of news articles. Red articles are conservative, blue are liberal. On the left is the optimal unfair hierarchy. We alter the hierarchy slightly on the right to achieve fairness. Now, the user's query for global warming will yield both liberal and conservative articles.

liberal-leaning articles in blue. We can see that in this example, when the user searches for global warming articles, they will only see liberal articles. To resolve this, we add a group fairness constraint on our cluster: for example, require at least 1/3 of the articles in each cluster to be of each political skew. This guarantees (as depicted on the right) that the outputted articles will always be at least 1/3 liberal and 1/3 conservative, thus guaranteeing the user is exposed to a range of perspectives along this political axis. This notion of fairness, which we formally define in Definition 81, has been explored in the context of hierarchical clustering in both [Ahmadian et al., 2020b] and [Knittel et al., 2023b].

This paper is concerned with approximations for fair low-cost (i.e., optimizing for [Dasgupta, 2016]'s famous cost metric) hierarchical clustering. This is perhaps the most natural and well-motivated metric for hierarchical clustering evaluation, however it is quite difficult to optimize for (the best being $O(\sqrt{\log n})$-approximations [Charikar and Chatziafratis, 2017, Dasgupta, 2016]; it hypothesized to not be $O(1)$-approximable [Charikar and Chatziafratis, 2017]). This appears to be even more difficult in the hierarchical clustering literature. The first work to attempt this problem, [Ahmadian et al., 2020b], achieved a highly impractical $O(n^{5/6}\log^{3/2} n)$-

approximation (not too far from the trivial $O(n)$ upper bound), posing fair low-cost hierarchical clustering as a in interesting and inherently difficult problem. [Knittel et al., 2023b] greatly improved this to a near-polylog approximation factor of $O(n^{\delta}\text{polylog }(n))$, where δ can be arbitrarily close to 0, and parameterizes a trade-off between approximation factor and degree of fairness. Still, a true polylog approximation was left as an open problem, one which we solve in this paper.

4.1.1 Our Contributions

This work proposes the first polylogarithmic approximation for fair, low-cost hierarchical clustering. We leverage the work of [Knittel et al., 2023b] as a starting inspiration and create something much simpler, more direct, and better in both fairness and approximation. Like their algorithm, our algorithm starts with a low-cost unfair hierarchical clustering and then alters it with multiple well-defined and limited tree operators. This gives it a degree of explainability, in that the user can understand exactly the steps the algorithm took to achieve its result and why. In addition, our algorithm achieves both relative cluster balance (i.e., clusters who are children of the same cluster have similar size) and fairness, along with a parameterizeable trade-off between the two.

On top of the benefits of [Knittel et al., 2023b]'s techniques, we propose a greatly simplified algorithm. They initially proposed an algorithm that required four tree operators, however, we only require two of the four, and we greatly simplify the more complicated operator. This makes the algorithm simpler to understand and more implementable. We show that even with this reduced functionality, we can cleverly achieve both a better approximation and degree of fairness:

Theorem 1. *When T is a γ-approximate low-cost vanilla hierarchical clustering over $\ell(V) = c_\ell n = O(n)$ vertices of each color $\ell \in [\lambda]$,* MakeFair *(Algorithm 11), for any constants ϵ, h, k with $h >> k^\lambda$ and $n >> h$, runs in $O(n \log n(h + \lambda \log n))$ time and yields a hierarchy T' satisfying:*

1. *T' is an $O\left(\frac{(h-1)}{\epsilon} + \frac{1+\epsilon}{1-\epsilon}k^\lambda\right)$-approximation for cost.*
2. *T' is fair for any parameters for all $i \in [\lambda]$: $\alpha_i \leq \frac{\lambda_i}{n}\left(\frac{1-\epsilon}{(1+\epsilon)^2}\left(1 - \frac{k(1+\epsilon)}{c_i h}\right)\right)^{O(\log(n))}$ and $\beta_i \geq \frac{\lambda_i}{n}\left(\frac{1+\epsilon}{(1-\epsilon)^2}\left(1 + \frac{1-\epsilon}{c_i k}\right)\right)^{O(\log(n))}$, where $\lambda_i = c_i n$.*
3. *All internal nodes in T' are ϵ-relatively balanced.*

To put this in perspective, previously, the best approximation for fair hierarchical clustering previously was $O(n^\delta \text{polylog}\,(n))$, whereas the best unfair approximation is $O(\sqrt{\log n})$. Our work greatly reduces this gap by providing a true $O(\text{polylog}\,(n))$ approximation. This can be achieved by setting $k = O(1)$, $h = O(\log n)$, and $\epsilon = O(1/\log n)$ (note we assume $\lambda = O(1)$ and the best current $\gamma = O(\sqrt{\log n})$:

Corollary 78. *There is a hierarchical clustering algorithm which runs in $O(n \log^2 n)$ time and yields a hierarchy T' satisfying: 1) T' is an $O(\log^2(n))$-approximation for cost, 2) T' is fair for any parameters for all $i \in [\lambda]$: $\alpha_i = a_i \frac{\lambda_i}{n}$ and $\beta_i \geq b_i \frac{\lambda_i}{n}$ where $a_i \in (0, 1)$ and $b_i > 1$ are constants for all $i \in [\lambda]$, and 3) All internal nodes in T' are $O\left(\frac{1}{\log n}\right)$-relatively balanced.*

4.2 Preliminaries

4.2.1 The Vanilla Problem

Fair clustering literature refers to the original problem variant, without fairness, as the "vanilla" problem. We define the vanilla problem of finding a low-cost hierarchical clustering here using our specific notation.

In this problem, we are given a complete graph $G = (V, E, w)$ with a weight function $w : E \to \mathbb{R}^+$ is a measure of the similarity between datapoints. Note the data is encoded as a complete tree because we require knowledge of all point-point relationships. We must construct a hierarchical clustering, represented by its dendrogram, T, with root denoted $\text{root}(T)$. T is a tree with vertices corresponding to all clusters of the hierarchical clustering. Leaves of T, denoted $\text{leaves}(\text{root}(T))$ correspond to the points in the dataset (i.e., singleton clusters). An internal node v corresponds to the cluster containing all leaf-data of the maximal subtree (i.e., contains all its descendants) rooted at v, $T[v]$. In addition, we let $u \wedge v$ denote the lowest common ancestor of u and v in T.

In order to define [Dasgupta, 2016]'s cost function, we use the same notational simplifications as [Knittel et al., 2023b]. For an edge $e = (x, y) \in E$, we say $n_T(e) = |\text{leaves}(T[x \wedge y])|$ is the size of the smallest cluster in the hierarchy containing e. Similarly, for a hierarchy node v, $n_T(v_i) = |\text{leaves}(T[v_i])|$ is the size of the corresponding cluster. This is sufficient to introduce the notion of *cost*.

Definition 79 ([Knittel et al., 2023b]). *The **cost** of $e \in E$ in a graph $G = (V, E, w)$ in a hierarchy T is* $\text{cost}_T(e) = w(e) \cdot n_T(e)$.

Dasgupta's cost function can then be written as a sum over the costs of all edges.

Definition 80 ([Dasgupta, 2016]). *The **cost** of a hierarchy T on graph $G = (V, E, w)$ is:*

$$\text{cost}(T) = \sum_{e \in E} \text{cost}_T(e)$$

Our algorithm begins by assuming we have some approximate vanilla hierarchy, T. That is, if OPT is the optimal hierarchy tree, then $\text{cost}(T) \leq \alpha \cdot \text{cost}(OPT)$ for some approximation factor α. According to [Dasgupta, 2016], we can transform this hierarchy to be binary without increasing cost. Our paper simply assumes our input is binary. We then produce a modified hierarchy T' which similar structure to T that guarantees fairness, i.e., $\text{cost}(T') \leq \alpha' \cdot \text{cost}(OPT)$ for some approximation factor $\alpha' \geq \alpha$. Note this comparison is being made to the vanilla OPT, as we are unsure, at this time, how to classify the optimal fair hierarchy. Note that the binary assumption may not hold when we consider adding a fairness constraint.

4.2.2 Fairness and Balance Constraints

We consider the fairness constraints based off those introduced by [Chierichetti et al., 2017] and extended by [Bercea et al., 2019]. On a graph G with colored vertices, let $\ell(C)$ count the number of ℓ-colored points in cluster C.

Definition 81 ([Knittel et al., 2023b]). *Consider a graph $G = (V, E, w)$ with vertices colored one of λ colors, and two vectors of parameters $\alpha, \beta \in (0, 1)^\lambda$ with $\alpha_\ell \leq \beta_\ell$ for all $\ell \in [\lambda]$. A hierarchy T on G is **fair** if for any non-singleton cluster C in T and for every $\ell \in [\lambda]$, $\alpha_\ell |C| \leq \ell(C) \leq \beta_\ell |C|$. Additionally, any cluster with a leaf child has only leaf children.*

Effectively, we are given bounds α_ℓ and β_ℓ for each color ℓ. Every non-singleton cluster must have at least an α_ℓ fraction and at most a β_ℓ fraction of color ℓ. This guarantees proportional representational fairness of each color in each cluster.

As an intermediate step in achieving fairness, we will create splits in our hierarchy that achieve relative balance in terms of subcluster size. Thus, the following definition will come in handy.

Definition 82. *In a hierarchy, a vertex v (corresponding to cluster C) with c_v children is ϵ-**relatively balanced** if for every cluster $\{C_i\}_{i\in[c_v]}$ that corresponds to a child of v, $(\frac{1}{c_v} - \epsilon)|C| \leq |C_i| \leq (\frac{1}{c_v} + \epsilon)|C|$.*

While this definition is quite similar to that from [Knittel et al., 2023b], it deviates in two ways: 1) we only define it on a single split as opposed to the entire hierarchy and 2) we allow splits to be non-binary. If we apply it to the entire hierarchy and constrain it to be binary, it is equivalent to the former definition.

4.2.3 Tree Operators

Our work simplifies the work of [Knittel et al., 2023b]. In doing so, we follow the same framework, using tree operators to make well-defined and limited alterations to a given hierarchical clustering (Figure 4.2). In addition, our algorithm simplifies operator use in two ways: 1) we only utilize two of their four tree operators, and 2) we greatly simplified their most complicated operator and show that it can still be used to create a fair hierarchy.

First off, we utilize the same subtree deletion and insertion operator. The main difference is how we use it, which will be discussed in Section 4.3. At a high level, this operator removes

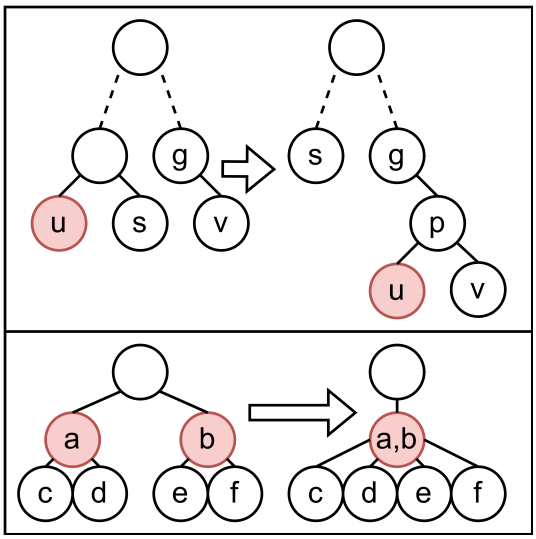


Figure 4.2: Our operators: subtree deletion and insertion and shallow tree folding.

a subtree from one part of the hierarchy and reinserts it elsewhere, adding and removing parent vertices as necessary.

Definition 83 ([Knittel et al., 2023b])**.** *Consider a binary tree T with internal nodes u, some non-ancestor v, u's sibling s, and v's parent g.* ***Subtree deletion*** *at u removes $T[u]$ from T and contracts s into its parent.* ***Subtree insertion*** *of $T[u]$ at v inserts a new parent p between v and g and adds u as a second child of p. The operator $\mathrm{del\_ins}(u, v)$ deletes u and inserts $T[u]$ at v.*

The other operator we leverage is their tree folding operator, however we greatly simplify it. In the previous work, tree folding took two or more isomorphic trees and mapped the internal nodes to each other. Instead, we simply take two or more subtrees and merge their roots. The new root then directly splits into all children of the roots of all folded trees. In a way, this is an implementation of their folding operator but only at a single vertex in the tree topology. This is why we call it a shallow tree fold.

Definition 84. *Consider a set of subtrees $T_1, \ldots, T_k$ of T such that all $\mathrm{root}(T_i)$ have the same parent p in T. A* ***shallow tree folding*** *of trees $T_1, \ldots, T_k$ ($\mathrm{fold}(T_1, \ldots, T_k)$) modifies T such that*

all $T_1, \ldots, T_k$ *are replaced by a single tree* T_f *whose* $\text{root}(T)$ *is made a child of* p*, and* $T_1, \ldots, T_k$ *make up the direct descendants of* $\text{root}(T_f)$.

In addition, we assume the subtree T_f is then arbitrarily binarized [Dasgupta, 2016] after folding. Since our algorithm works top-bottom, creating balanced vertices as it goes, we don't yet care about the fairness of the descendants of T_f. Moreover, we will then recursively call our algorithm on T_f to do precisely this.

4.3 Main Algorithm

In this section, we present our fair, low-cost, hierarchical clustering algorithm along with its analysis. Ultimately, we achieve the following (for a more intuitive explanation, see Section 4.1):

Theorem 1. *When* T *is a* γ*-approximate low-cost vanilla hierarchical clustering over* $\ell(V) = c_\ell n = O(n)$ *vertices of each color* $\ell \in [\lambda]$*,* MakeFair *(Algorithm 11), for any constants* ϵ, h, k *with* $h >> k^\lambda$ *and* $n >> h$*, runs in* $O(n \log n(h + \lambda \log n))$ *time and yields a hierarchy* T' *satisfying:*

1. T' *is an* $O\left(\frac{(h-1)}{\epsilon} + \frac{1+\epsilon}{1-\epsilon}k^\lambda\right)$*-approximation for cost.*
2. T' *is fair for any parameters for all* $i \in [\lambda]$*:* $\alpha_i \leq \frac{\lambda_i}{n}\left(\frac{1-\epsilon}{(1+\epsilon)^2}\left(1 - \frac{k(1+\epsilon)}{c_i h}\right)\right)^{O(\log(n))}$ *and* $\beta_i \geq \frac{\lambda_i}{n}\left(\frac{1+\epsilon}{(1-\epsilon)^2}\left(1 + \frac{1-\epsilon}{c_i k}\right)\right)^{O(\log(n))}$*, where* $\lambda_i = c_i n$.
3. *All internal nodes in* T' *are* ϵ*-relatively balanced.*

The main idea of our algorithm is to leverage similar tree operators to that of [Knittel et al., 2023b], but greatly simplify their usage and apply them in a more direct, careful manner. Specifically, the previous work processes the tree four times: once to achieve $1/6$-relative balance

everywhere, next to achieve ϵ-relative balance, next to remove the bottom of the hierarchy, and finally to achieve fairness. The problem is that this causes proportional cost increases to grow in an exponential manner, particularly because the relative balance significantly degrades as you descend the hierarchy. Our solution is to instead do a single top to bottom pass of the tree, rebalancing and folding to achieve fairness as we go. We describe this in detail now.

First, we assume our input is some given hierarchical clustering tree. Ideally, this will be a good approximation for the vanilla problem, but our results do work as a black box on top of any hierarchical clustering algorithm. Second, we apply $\mathrm{SplitRoot}$ in order to balance the root (Section 4.3.1). And finally, we apply shallow tree folding on the children of the root to achieve fairness (Section 4.3.2). This gives us the first layer of our output, and then we recurse.

4.3.1 Root Splitting and Balancing

$\mathrm{SplitRoot}$ is depicted in Algorithm 10. This fills the role of [Knittel et al., 2023b]'s Refine Rebalance Tree algorithm (and skips their Rebalance Tree algorithm), but it functions differently in that it only rebalances the root and it immediately splits the root into h children, according to our input parameter h.

We start $\mathrm{SplitRoot}$ by adding dummy children to v until it has h children (recall we can assume the input is binary). A dummy or null child is just a placeholder for a child to be constructed, or alternatively simply a zero-sized tree (note: this does not add any leaves to the tree). None of these children will be left empty in the end. Next, we define v_{max} and v_{min}, the maximal subtrees rooted at $\mathrm{children}(\mathrm{root}(T'))$ which have the most and fewest leaves, respectively.

As long as the root is not ϵ-relatively balanced (which is equivalent to $n_{T'}(v_{max})$ or $n_{T'}(v_{min})$

deviating from the target n/h by over $n\epsilon$, as they are extreme points), we will attempt to rebalance. We define δ_1 and δ_2 to be the proportional deviation of $n_{T'}(v_{min})$ and $n_{T'}(v_{max})$ from the target size n/h respectively, and δ to be the minimum of the two. In effect, δ measures the maximum number of leaves we can move from the large subtree to the small subtree without causing $n_{T'}(v_{max})$ to dip below n/h or $n_{T'}(v_{min})$ to peak above n/h. This is important to guarantee our runtime: as an accounting scheme, we show that clusters monotonically approach size n/h, and thus we can quantify how fast our algorithm completes. We fully analyze this later, in Lemma 86.

Algorithm 10 SplitRoot

Input A binary hierarchy tree T of size $n \geq 1/2\epsilon$ over a graph $G = (V, E, w)$, with smaller cluster always on the left, and parameters $h \in [n]$ and $\epsilon \in (0, \min(1/6, 1/h))$.

Output A hierarchical clustering T' with an ϵ-relatively balanced root that has k children.

Initialize $T' = T$
$v = \text{root}(T')$
Add null children to v until it has h children
Let $v_{min} = \text{argmin}_{v' \in \text{children}(v)} n_{T'}(v')$
Let $v_{max} = \text{argmax}_{v' \in \text{children}(v)} n_{T'}(v')$
while $n_{T'}(v_{max}) > n(1/h + \epsilon)$ or $n_{T'}(v_{min}) < n(1/h - \epsilon)$ **do**
    $\delta_1 = 1/h - n_{T'}(v_{min})/n$
    $\delta_2 = n_{T'}(v_{max})/n - 1/h$
    $\delta = \min(\delta_1, \delta_2)$
    Let $v = v_{max}$

    while $n_{T'}(v) > \delta n$ **do**
        $v \leftarrow \text{right}_{T'}(v)$
    end while

    $u \leftarrow v_{min}$
    while $n_{T'}(\text{right}_{T'}(u)) \geq n_{T'}(v)$ **do**
        $u \leftarrow \text{left}_{T'}(u)$
    end while
    $T' \leftarrow T'.\text{del\_ins}(u, v)$
    Reset v_{min} and v_{max}
end while

Now we must attempt exactly this: move a large subtree from v_{max} to v_{min}, though this subtree can have no more than δn leaves. To do this, we simply start at v_{max} and traverse down

its right children (recall below v_{max}, the tree is still binary). We halt on the first child that is of size δn or smaller. We then remove it and find a place to reinsert it under v_{min}.

The insertion spot is found similarly by descending down v_{min}'s left children until the right child of the current vertex has fewer leaves in its subtree than the tree we are inserting. Thus we have completed our insertion/deletion operation. We repeat until the tree is relatively balanced, as desired.

We now analyze this part of the algorithm. The full proofs can be found in the appendix, but we give intuition here. To start, consider the tree we are deleting and reinserting, $T'[v]$. Ideally, we want this to have many leaves, but no more than δn. We find that:

Lemma 85. *For a subtree $T'[v]$ that is deleted and reinserted in* SplitRoot *(Algorithm 10), $\epsilon n/(2(h-1)) < n_T(v) \leq \delta n$.*

The upper bound simply comes from our stopping condition in the first nested while loop: we ensure $n_{T'}(v) \leq \delta n$ before selecting it. The lower bound is slightly more complicated. Effectively, we start by noting that $\max(\delta_1, \delta_2) > \epsilon$, because otherwise the stopping condition for the outer loop would be met. Then, consider the total amount of "excess of large clusters", or more precisely, the sum over all deviations from n/h of clusters larger than n/h (note if all clusters were n/h, it would be perfectly balanced). This total excess must be matched in the "deficiency of small clusters", which is the sum of deviations of clusters smaller than n/h. Therefore, since there are at least h small or h large clusters, the largest deviation must be at most h times the smallest deviation, according to our accounting scheme. This allows us to bound $\delta \geq \epsilon/(h-1)$. The tree that is inserted and deleted must have at least half this many leaves, since it is the larger child of a node with over δn leaves in its subtree. This gives our lower bound,

showing we move at least a significant number of vertices each step.

Next, we want to show the relative balance. Along with the analysis, we also get the runtime, which turns out to be near linear, assuming $h \ll n$.

Lemma 86. SplitRoot *(Algorithm 10) yields a hierarchy whose root is ϵ-relatively balanced with h children. In addition, it requires $O(nh)$ time to halt.*

To see why this is true, it's pretty obvious the root has h children, as this is set at the beginning and never changes. The runtime comes from our aforementioned accounting scheme: the total excess and deficiency is reduced by the number of leaves in the subtree we move at each step, which we showed in Lemma 85 is $n\epsilon/(2(h-1))$ at least. This gives us a convergence time of $O(h)$, and each step can be bounded by $O(n)$ time as we search for our insertion and deletion spots. Finally, the balance comes from the fact that our stopping condition is equivalent to the root being relatively balanced.

All that is left is to show the negative impact on the cost of edges that are separated by the algorithm. We bound it as follows:

Lemma 87. *In* SplitRoot *(Algorithm 10), for all $e \in E$ that is separated:*

$$\text{cost}_{T'}(e) \leq n \cdot w(e) \leq 2(h-1) \cdot \text{cost}_T(e)/\epsilon$$

Lemma 85 tells us that moved subtrees are at least of size $\epsilon n/(2(h-1))$, which is a lower bound on the size of the smallest cluster containing any edge separated by the algorithm. This is because separated edges must have one endpoint in the deleted subtree and one outside, so their least common ancestor is an ancestor of the subtree. At worst, the final size of the smallest cluster

containing such an edge is n, so the proportional increase is $2(h-1)/\epsilon$ at worst.

4.3.2 Fair Tree Folding

Next, we discuss how to achieve fairness by using $\mathrm{MakeFair}$, as seen in Algorithm 11. This is our final recursive algorithm which utilizes $\mathrm{SplitRoot}$. Assume we are given some hierarchical clustering. We start by running $\mathrm{SplitRoot}$, to balance the split at the root and give it h children. Next we use a folding process similar to that of [Knittel et al., 2023b], but we use our shallow tree fold operator.

More specifically, we first sort the children of the root by the proportional representation of the first color (say, red). Then, we do a shallow fold across various k-sized sets, defined as follows: according to our ordering over the children, partition the vertices into k contiguous chunks starting from the first vertex. For each $i \in [h/k]$, we find the ith vertex in each chunk and fold them together. Notice that this is a k-wise fold since there are k chunks, and we end up with h/k vertices. This is repeated on each color. After this, we simply recurse on the children. If a child is too small to be balanced by $\mathrm{SplitRoot}$, then we stop and give it a trivial topology (a root with many leaf-children).

This completes our algorithm description. We now evaluate its runtime, degree of fairness, and approximation factor. To start, we show the degree of fairness achieved at the top level of the hierarchy.

Lemma 88. $\mathrm{MakeFair}$ *(Algorithm 11) yields a hierarchy such that all depth 1 vertices satisfy fairness under* $\alpha_i \leq \frac{\lambda_i}{n} \cdot \frac{1-\epsilon}{(1+\epsilon)^2}\left(1 - \frac{k(1+\epsilon)}{c_i h}\right)$ *and* $\beta_i \geq \frac{\lambda_i}{n} \cdot \frac{1+\epsilon}{(1-\epsilon)^2}\left(1 + \frac{1-\epsilon}{c_i k}\right)$*, where* $\lambda_i = c_i n$*.*

This proof is quite in depth, and most details are deferred to the appendix. At a high level,

we are showing that the folding process guarantees a level of fairness. The parts in our partition are ordered by the density of the color (say, red). Since each final vertex is made by folding across one vertex in each part, meaning that the vertices have a relatively wide spread in terms of their density of red poitns. This means that red vertices are distributed relatively well across our final subtrees. This guarantees a degree of balance.

The problem is that the degree of fairness still exhibits a compounding affect as we recurse. That is, since the first children are not perfectly balance, then in the next recursive step, the total data subset we are working on may now deviate from the true color proportions. This deviation is bounded by our result in Lemma 88, but it will increase proportionally at each step.

Lemma 89. *In* MakeFair *(Algorithm 11), let* $\{\lambda_i\}_{i\in[\lambda]}$ *be the proportion of each color and assume* $k^\lambda << h$*. At any recursive call, the proportion of any color is (where* $\lambda_i = 1/c_i$ *for constant* c_i*):*

$$\lambda_i \left(\frac{1-\epsilon}{(1+\epsilon)^2}\left(1-\frac{k(1+\epsilon)}{c_i h}\right)\right)^{O(\log(n/h))} \leq \lambda_i^j \leq \lambda_i \left(\frac{1+\epsilon}{(1-\epsilon)^2}\left(1+\frac{1-\epsilon}{c_i k}\right)\right)^{O(\log(n/h))}$$

Also, the recursive depth is bounded above by $O(\log(n/h))$.

This fairly neatly comes from Lemma 88. Effectively, we increase the proportion of each color by the same factor each recursive step. All that is left to do is bound the recursive depth. Notice we start with n vertices. After splitting, our subtrees have size at most $(1+\epsilon)n/h$. After one fold, this is increased by a factor of k, and thus k^λ after all folds. Interestingly, this doesn't impact the final result significantly; it's fairly similar to turning an n-sized tree into an n/h-sized tree, giving an $O(\log(n/h))$ recursive depth. This will be sufficient to show our fairness.

Algorithm 11 MakeFair

Input A hierarchy tree T of size $n \geq 1/2\epsilon$ over a graph $G = (V, E, w)$ with vertices given one of λ colors, and parameters $h \in [n]$, $k \in [h/(\lambda - 1)]$, and $\epsilon \in (0, \min(1/6, 1/h))$.

Output A fair hierarchical clustering T'.

$T' = \text{SplitRoot}(T, h, \epsilon)$
$h' \leftarrow h$
for each color $\ell \in [\lambda]$ **do**
    Order $\{v_i\}_{i \in [h']} = \text{children}(\text{root}(T'))$ decreasing by $\frac{\ell(\text{leaves}(v_i))}{n_{T'}(v_i)}$
    For all $i \in [k]$, $T' \leftarrow T'.\text{fold}(\{T'[v_{i+(j-1)k}] : j \in [h'/k]\})$
    $h' \leftarrow h'/k$
end for
for each child v_i of $\text{root}(T')$ **do**
    if $n \geq \max(1/2\epsilon, h)$ **then**
        Replace $T'[v_i] \leftarrow \text{MakeFair}(T'[v_i], h, k, \epsilon)$
    else
        Replace $T'[v_i]$ with a tree of root v_i, leaves $\text{leaves}(T'[v_i])$, and depth 1.
    end if
end for

Next, we evaluate the cost incurred at each stage in the hierarchy.

Lemma 90. *In* MakeFair *(Algorithm 11), for all $e \in E$ that is separated before the recursive call:*

$$\text{cost}_{T'}(e) \leq O\left(\frac{2(h-1)}{\epsilon} + \frac{1+\epsilon}{1-\epsilon}k^{\lambda}\right)\text{cost}_{T}(e)$$

As discussed before, the final cluster size should be $(1+\epsilon)nk^{\lambda}/h$. Any separated edge must have a starting cluster size of at least $(1-\epsilon)n/h$, as this is the size of the smallest cluster involved in tree folding. From this, it is simple to compute the proportional cost increase of a single recursive level. We must also account for the cost increase from the initial splitting, from Lemma 87.

Another nice property of our method is that whenever an edge is separated, its endpoints' least common ancester will no longer be involved in any further recursive step. This tells us:

Lemma 91. *In* MakeFair *(Algorithm 11), any edge $e \in E$ is separated at only one level of*

recursion.

Putting these two together pretty directly gives us our cost approximation.

Lemma 92. *In* MakeFair *(Algorithm 11),* $\text{cost}(T') \leq O\left(\frac{2(h-1)}{\epsilon} + \frac{1+\epsilon}{1-\epsilon}k^{\lambda}\right)\text{cost}(T)$.

Finally, Theorem 1 comes directly from Lemmas 90 and 92.

4.4 Simulations

This section validates the theoretical guarantees of Algorithm 11. Specifically, we demonstrate that modifying an unfair hierarchical clustering using the presented procedure yields a fair hierarchy that incurs only a modest increase in cost.

Datasets. We use two data sets, *Census* and *Bank*, from the UCI data repository [Dua and Graff, 2017]. Within each, we subsample only the features with numerical values. To compute the *cost* of a hierarchical clustering we set the similarity to be $w(i,j) = \frac{1}{1+d(i,j)}$ where $d(i,j)$ is the Euclidean distance between points i and j. We color data based on binary (represented as blue and red) protected features: *race* for *Census* and *marital status* for *Bank* (both in line with the prior work of [Ahmadian et al., 2020b]). As a result, *Census* has a blue to red ratio of 1:7 while *Bank* has 1:3. We then subsample each color in each data set such that we retain (approximately) the data's original balance. We use samples of size 512 for the balance experiments, and vary the sample sizes when assessing cost. For each experiment we conduct 10 independent replications (with different random seeds for the subsampling), and report the average results. We vary the parameters (c, h, k, ε) to experimentally assess their theoretical impact on the approximate guarantees of Section 4.3. Due to space constrains, we here present only the results for the *Census* dataset and defer the complimentary results on *Bank* to the appendix.

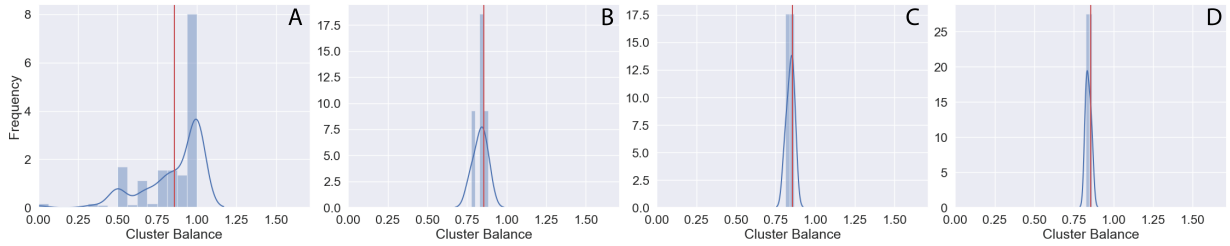


Figure 4.3: Histogram of cluster balances after tree manipulation by Algorithm 11 on a subsample from the *Census* dataset of size $n = 512$. The four panels depict: **(A)** cluster balances after applying the (unfair) average-linkage algorithm, **(B)** the resultant cluster balances after running Algorithm 11 with parameters $(c, h, k, \varepsilon) = (8, 4, 2, 1/c \cdot \log_2 n)$, **(C)** cluster balances after tuning $c = 4$, **(D)** cluster balances after further tuning $c = 2$. The vertical red line on each plot indicates the balance of the dataset itself.

Implementation. The Python code for the following experiments are available in the Supplementary Material. We start by running average-linkage, a popular hierarchical clustering algorithm. We then apply Algorithm 11 to modify this structure and induce a *fair* hierarchical clustering that exhibits a mild increase in the cost objective.

Metrics. In our results we track the approximate cost objective increase as follows: Let G be our given graph, T be average-linkage's output, and T' be Algorithm 11's output. We then measure the ratio $\text{RATIO}_{cost} = cost_G(T')/cost_G(T)$. We additionally quantify the fairness that results from application of our algorithm by reporting the balances of each cluster in the final hierarchical clustering, where true fairness would match the color proportions of the underlying dataset.

Results. We first demonstrate how our algorithm adapts an unfair hierarchy into one that achieves fair representation of the protected attributes as desired in the original problem formulation.

In Figure 4.3, we depict the cluster balances of an *unfair* hierarchical clustering algorithm, namely "average-linkage", and subsequently demonstrate that our algorithm effectively concentrates all clusters around the underlying data balance. In particular, we first apply the algorithm

and then show how we the balance is further refined by tuning the parameters. The application of Algorithm 11 dramatically improves the representation of the protected attributes in the final clustering and, as such, firmly resolves the problem of achieving fairness.

While reaching this fair partitioning of the data is the overall goal, we further demonstrate that, in modifying the unfair clustering, we only increase the cost approximation by a modest amount. Figure 4.4 illustrates the change in relative cost as we increase the sample size n, the primary influence on our theoretical cost guarantees of Section 4.3. Specifically, we vary n in $\{128, 256, 512, 1024, 2048\}$ and compute 10 replications (on different random seeds) of the fair hierarchical clustering procedure. Figure 4.4 depicts the mean relative cost of these replications with standard error bars. Notably, we see that the cost does increase with n as expected, but the increase relative to the unfair cost obtain by average linkage is only by a small multiplicative factor.

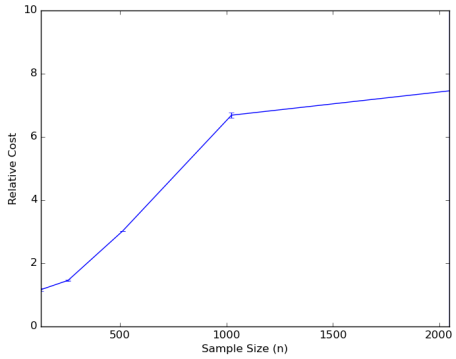


Figure 4.4: Relative cost of the fair hierarchical clustering resulting from Algorithm 11 compared to the unfair clustering as a function of the sample size n.

As demonstrated through this experimentation, the simplistic procedure of Algorithm 11 not only ensures the desired fairness properties absent in conventional (unfair) clustering algorithms but accomplishes this feat with a negligible rise in the overall cost. These results further highlight the immense value of our work.

4.5 Limitations

Fair machine learning strives to combat the limitations of vanilla machine learning by providing a means for bias mitigation for any desired quantifiable bias. However, fair research itself has its own limitations. First, "fairness" can be defined in a number of ways. For instance, [Dwork et al., 2012] explores notions of fairness in classification problems, proposing a type of "individual fairness" which guarantees that similar individuals are treated similarly. This has been extended to clustering by only the work of [Brubach et al., 2020]. Clustering has been predominantly viewed through the lens of "group fairness" which guarantees that different protected classes receive similar, proportional treatment. This was first proposed in clustering by [Chierichetti et al., 2017] and expanded upon in many further works [Ahmadian et al., 2019, Bera et al., 2019, Bercea et al., 2019], including previous fair hierarchical clustering work [Ahmadian et al., 2020b, Knittel et al., 2023b] and this work. Not only is it inherently difficult to account for both of these simultaneously, in some sense these two notions are at odds: if we treat similar individuals similarly, it becomes much harder to impose a diverse range of treatments to individuals in each group, as they often are quite similar themselves. This illustrates the necessity of applying fair algorithms on a case by case basis, carefully considering what fair effect is most desirable.

Second, bias mitigation through fair algorithmic techniques has been shown to cause harm in at least one application [Ben-Porat et al., 2021]. Thus, all fair machine learning techniques, including ours, should be used with great caution and consideration of all downstream effects. We defer the reader to [Barocas et al., 2019] as well as the Fair Clustering Tutorial [AAAI 2023] for further perspectives on fair machine learning and its limitations.

The main results of this paper are theoretical guarantees on algorithmic performance. Naturally, this provides additional limitations, predominantly in that the guarantees only hold under the assumptions clearly stated in this paper. For instance, our main algorithm requires that each color represents a constant fraction of the total data. This assumption is quite realistic and can be found throughout fair learning literature, but there are certain practical instances where our results may not be applicable. In addition, since our proofs only consider worst-case analysis, we do not know much about the average-case guarantees of our algorithms (other than they are strictly better than the worst case). We account for this through empirical evaluation, though this is inherently limited as tested data sets cannot represent all potential applications.

Finally, our work focuses on the cost objective function. While cost is highly regarded by the hierarchical clustering community [Dasgupta, 2016], it may not be an appropriate metric for all applications. Moreover, it is sometimes viewed as impractical in that it is quite difficult to provide worst-case guarantees for [Charikar and Chatziafratis, 2017]. Future work might consider evaluating our algorithms using other objectives such as revenue [Moseley and Wang, 2017] or value [Cohen-Addad et al., 2018] to see how they perform.

4.6 Proofs

This section contains the formal proofs for all of our lemmas and theorems.

Proof of Lemma 85. We start by comparing δ and ϵ at some iteration. Consider v_{min} and v_{max} at that iteration. Without loss of generality, say $n/h - n_{T'}(v_{min})/n \leq n_{T'}(v_{max})/n - n/h$, implying $\delta = \delta_1 = n/h - n_{T'}(v_{min})/n$. Additionally, since the while loop executed, we know either $n_{T'}(v_{max}) = n(1/h + \delta_2) > n(1/h + \epsilon)$ or $n_{T'}(v_{min}) = n(1/h - \delta_1) < n(1/h - \epsilon)$. With

a little algebraic simplification, this gives us that $\delta_1 > \epsilon$ or $\delta_2 > \epsilon$. Since we said $\delta = \delta_1$, δ_1 must be the smaller, so we can safely assume $\delta_2 > \epsilon$.

Now, we know conservatively that $\delta_2 < n_{T'}(v_{max})/n \leq 1$. Since $n_{T'}(v_{min})/n$ has the largest deviation from $1/h$ of all of $v' \in \text{children}(\text{root}(T'))$ with $n_{T'}(v') \leq n/h$, this means that $1/h - n_{T'}(v')/n \leq \delta_1$ for all $v' \in \text{children}(\text{root}(T'))$, in other words, $n_{T'}(v') \geq n(1/h - \delta_1)$. Since $\text{children}(\text{root}(T'))$ form a clustering of the data, $\sum_{v' \in \text{children}(\text{root}(T'))} n_{T'}(v') = n$. In addition, because of our bound:

$$\begin{aligned}
\sum_{v' \in \text{children}(\text{root}(T'))} n_{T'}(v') &= \sum_{v' \in \text{children}(\text{root}(T')) \setminus v_{max}} n_{T'}(v') + n_{T'}(v_{max}) \\
&\geq (h-1) \cdot n(1/h - \delta_1) + n/h + n\delta_2 \\
&= n - n(h-1)\delta_1 + n\delta_2
\end{aligned}$$

Recall our original value is n. Thus $n \geq n - n(h-1)\delta_1 + n\delta_2$. Finally, we get $\delta_1 \geq \delta_2/(h-1)$. This means $\delta \geq \epsilon/(h-1)$. A similar math can show the same result if δ_2 is the smaller value. For an upper bound, we have that since the smallest cluster size is 0, $\delta \leq \delta_1 \leq 1/h$.

Let p be the parent of v. By the halting condition of the while loop on Line 13, we know $n_{T'}(p) > \delta n$, otherwise the loop would have halted earlier. Since v is the right child of p, it is the larger of two children, implying $n_{T'}(v) \geq n_{T'}(p)/2 > \delta n/2$, which is just at least $\epsilon/(2(h-1))$ by our previous math. Finally, since the loop did halt on v, we know $n_{T'}(v) \leq \delta n$. □

Proof of Lemma 86. First off, clearly the root has h children, because we give it h children and never change this.

For the runtime, notice that we always decrease the number of leaves of the child with the max number of leaves. Let $n_{tot} = \sum_{v' \in \text{children} v : n_{T'}(v') > 1/h} n_{T'}(v') - n/h \leq n$. Note that the number of vertices in this summation is only ever reduced, since we swap at most δn vertices from the largest to the smallest vertex, implying the smallest vertex will never exceed n/h. Since v_{max} is necessarily involved in this sum (if not, then $n_{T'}(v_{max}) = 1/h$, implying all children are of equal size, meaning the algorithm already halted), and $n_{T'}(v_{max})$ is reduced by at least $\epsilon n/(2(h-1))$ each iteration by Lemma 85, we require at most $2(h-1)$ iterations of the while loop before we halt. In each iteration, we traverse down two subtrees to delete and insert, which takes at most $O(n)$ time each, for a total of $O(nh)$ time to complete the algorithm.

Finally, assume for contradiction it is not ϵ-relatively balanced with respect to h children. This means that in the output, either: 1) some vertex has under $(1/h - \epsilon)n$ leaves in its subtree, or 2) some vertex has over $(1/h + \epsilon)n$ leaves in its subtree. In the first case, this means $n_{T'}(v_{min}) < (1/h - \epsilon)n$, implying the while loop will continue to execute, contradicting that this is the resulting output. A similar argument holds in the second case. Thus, the root is ϵ-relatively balanced. □

Proof of Lemma 87. Consider an edge $e = (x, y)$ that is separated when we delete and insert. This can only happen if, without loss of generality, x is in the deleted/inserted component and y is not. Recall v whose subtree is deleted and reinserted. By Lemma 85, $n_T(v) > \epsilon n/(2(h-1))$.

Since x is a descendant of v and y is not, their lowest common ancestor v' must be an ancestor of v. Thus $n_T(v') > n_T(v) > \epsilon n/(2(h-1))$. Thus, $\text{cost}_T(e) = n_T(v') \cdot w(e) \geq n\epsilon \cdot w(e)/(2(h-1))$. In the end, the maximum cost is $\text{cost}_{T'}(e) \leq n \cdot w(e)$, therefore $\text{cost}_{T'}(e) \leq \frac{2(h-1)}{\epsilon}$. This concludes the proof.

□

Proof of Lemma 88. For simplicity, assume $k^\lambda | h$. Our algorithm first orders the depth 1 vertices decreasing by the fractional representation of the first color, say red. It then partitions it into parts of size h/k according to this order and folds all vertices of the same index in their part together. That is, k clusters are merged. We begin with h vertices, but after the $(x-1)$th fold, we only have h/k^x remaining. Let x be the iteration we are at in the folding process.

Let $f(i,j)$ denote the ith index in the jth partition of $\mathcal{V}$, i.e., $f(i,j) = jh/k + i$. Then for every $i \in [h/k]$, we create a new vertex u_i by folding $v_{f(i,j)}$ together for all $j \in k$. Let r_i denote the number of red vertices in u_i. For any i:

$$r_i/n_{T'}(u_i) = \frac{1}{n_{T'}(u_i)} \sum_{j \in [k]} red(v_{f(i,j)}) \leq \frac{1}{n_{T'}(u_i)} red(v_{f(1,1)}) + \frac{1}{n_{T'}(u_i)} \sum_{j \in \{2,\ldots,k\}} red(v_{f(i,j)})$$

Note that if we perfectly balanced all cluster sizes at n/h, then $red(v_{f(1,1)}) \leq n/h = n_{T'}(u_i)/k$ would hold. However, $v_{f(1,1)}$ may be a factor of at most $1+\epsilon$ larger and $n_{T'}(u_i)$ may be a factor of at least $1-\epsilon$ smaller. This means that our first term simplifies to $\frac{1+\epsilon}{k(1-\epsilon)}$.

For our second term, we note that $red(v_{f(i,j)})/n_T(v_{f(i,j)}) \leq red(v_{f(i,j-1)})/n_T(v_{f(i,j-1)})$. Since we have relative balance, all n_T values are within a factor of $\frac{1+\epsilon}{1-\epsilon}$ of each other. This means $red(v_{f(i,j)}) \leq \frac{1+\epsilon}{1-\epsilon} red(v_{f(i',j-1)})$ for all $i' \in [h/k]$. We can also take this as an average, as in, $red(v_{f(i,j)}) \leq \frac{k(1+\epsilon)}{h(1-\epsilon)} \sum_{i' \in [h/k]} red(v_{f(i',j-1)})$. Conservatively, this results in the summation $\sum_{j \in \{2,\ldots,h/k\}} \sum_{i' \in [k]} red(v_{f',j-1)})$. Here, we are practically counting (actually slightly undercounting) the total number of reds, which we call R. Plugging all of this in:

$$r_i/n_{T'}(u_i) \leq \frac{1+\epsilon}{k(1-\epsilon)} + \frac{(1+\epsilon)}{n(1-\epsilon)^2}R = \frac{R}{n} \cdot \frac{1+\epsilon}{(1-\epsilon)^2}\left(1 + \frac{1-\epsilon}{c_R k}\right)$$

Where since $R = O(n)$, we let c_R be the constant satisfying $R \geq c_R n$.

All that is left is to consider the lower bound. We can apply similar simplifications as before, but now we reverse the bound.

$$r_i/n_{T'}(u_i) = \frac{1}{n_{T'}(u_i)} \sum_{j \in [k]} red(v_{f(i,j)}) \geq \frac{1-\epsilon}{nh(1+\epsilon)^2} \sum_{j \in [k-1]} \sum_{i' \in [k]} red(v_{f(i',j+1)})$$

Again, we are undercounting R in the nested summations, though it is more problematic in the lower bound. Our missing terms are $\sum_{i' \in [k]} red(v_{f(i',1)})$. We can only bound this by the total size of the first partition, which is at most $(1+\epsilon)kn/h$.

$$r_i/n_{T'}(u_i) \geq \frac{1-\epsilon}{n(1+\epsilon)^2}\left(R - (1+\epsilon)kn/h\right) = \frac{R}{n} \cdot \frac{1-\epsilon}{(1+\epsilon)^2}\left(1 - \frac{k(1+\epsilon)}{c_R h}\right)$$

□

Proof of Lemma 89. We prove this inductively, saying at the jth level of recursion, $\lambda_i \left(\frac{1-\epsilon}{(1+\epsilon)^2}\left(1 - \frac{k(1+\epsilon)}{c_i h}\right)\right)^j \leq \lambda_i^j \leq \lambda_i \left(\frac{1+\epsilon}{(1-\epsilon)^2}\left(1 + \frac{1-\epsilon}{c_i k}\right)\right)^j$. This is obviously true in the base call to the algorithm, since $\lambda_i' = \lambda_i$. Assume this holds for level j.

In level $j+1$, any instance of the problem is really a subproblem on the hierarchy induced

on a cluster from the jth level of recursion. In that level of recursion, the number of vertices of color i, our induction shows that $\lambda_i \left(\frac{1-\epsilon}{(1+\epsilon)^2}\left(1 - \frac{k(1+\epsilon)}{c_i h}\right)\right)^j \leq \lambda_i^j \leq \lambda_i \left(\frac{1+\epsilon}{(1-\epsilon)^2}\left(1 + \frac{1-\epsilon}{c_i k}\right)\right)^j$. By Lemma 88, we can bound how much worse this gets by an additional multiplicative factor, yielding the desired inductive proof.

All that is left is to show the depth. At any recursive level, we begin with clusters of size of at most $(1+\epsilon)n/h$ after balancing. We fold k vertices together at most λ times, for a total size of at most $(1+\epsilon)nk^\lambda/h$. This means after the jth iteration, we have $n((1+\epsilon)k^\lambda/h)^j$ vertices left. Once we have only h vertices left, we will certainly stop. With a little simple arithmetic, we find this occurs when $j \leq \frac{\log(n/h)}{\log(h/((1+\epsilon)k^\lambda))} = O(\log(n/h))$ as long as $h \geq (1+\epsilon)k^\lambda$. This is the maximum number of iterations we require. Plugging this into our inductive finding gives the complete proof. □

Proof of Lemma 90. We already know that an edge e may be separated by $\mathrm{SplitRoot}$, and if so, it incurs a cost of $2(h-1)/\epsilon$. If this occurs, note that we already consider the worst case scenario: when $cost_{T'}(e) = n \cdot w(e)$. Therefore, if an edge is involved in separation in $\mathrm{MakeFair}$, the cost increase estimate cannot get worse.

We now consider an edge e that is separated in $\mathrm{MakeFair}$. It is not too hard to see that the cluster containing e must have been one of the depth 1 clusters, because otherwise e would not be affected by the algorithm. Therefore, $n_T(e) \geq (1-\epsilon)n/h$ (again, assuming it was not affected by the balancing). In the end, the max cluster size e belongs to will be $(1+\epsilon)nk^\lambda/h$, thus incurring a total cost increase of $\frac{1+\epsilon}{1-\epsilon}k^\lambda$. □

Proof of Lemma 91. This is not too hard to see. If an edge e is separated in a recursive level, that means the new worst-case ancestor is either the root at that level of recursion or the next. In the

former case, e is not involved in any further trees in the recursive process. In the latter case, it is contained in the root of one more recursive process. As this is already the most costly way to cluster e in the subproblem, it cannot be further separated. □

Proof of Lemma 92. This simply follows from Lemmas 90 and 91. The former shows the cost of separating an edge at a recursive level, and the latter says that this happens at most once to each edge. □

Proof of Theorem 1. Relative balance holds because we create relative balance in $\mathrm{SplitRoot}$. While we do fold these nodes together, merging nodes does not break relative balance. Our approximation factor is proved in Lemma 92. Lemma 89 gives us a bound on the proportion of each color in each recursive level, which in effect also tells us the actual fairness of each cluster in the hierarchy (i.e., by looking at the proportion of a certain color when we recurse on a cluster's subtree). This yields the desired fairness guarantee.

Finally, we showed the runtime for $\mathrm{SplitRoot}$ is $O(n'h)$ in Lemma 86, where n' is the current tree size. In $\mathrm{MakeFair}$, we require simple iteration and sorting to process the colors, and folding is a pretty simple process. Thus the first for loop only requires $O(n' \log n')$ time per execution for a total of $O(\lambda n' \log n')$ time. At any recursive level, a node is involved in at most one recursive instance. This means that the total time to execute a single recursive level is $O(n(h + \lambda \log n))$. Finally, Lemma 89 also tells us the recursive depth is bounded by $O(\log(n/h)) = O(\log n)$. Thus the total runtime is $O(n \log n(h + \lambda \log n))$. □

4.7 Additional Experiments

We here demonstrate how our algorithm adapts an unfair hierarchy into one that achieves fair representation of the protected attributes on the *Bank* dataset through a complimentary simulation to that of Section 4.4.

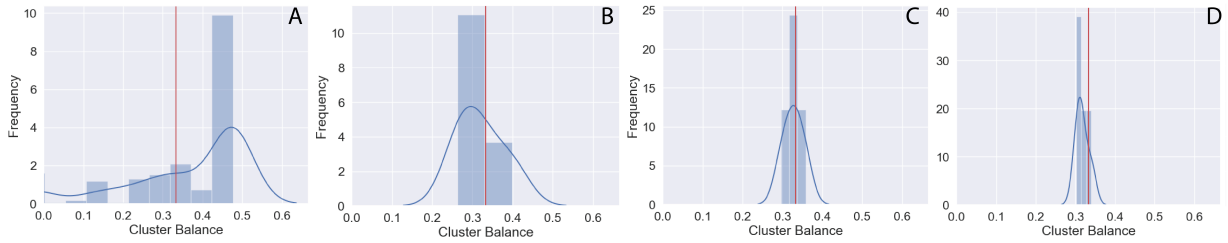


Figure 4.5: Histogram of cluster balances after tree manipulation by Algorithm 11 on a subsample from the *Bank* dataset of size $n = 512$. The four panels depict: **(A)** cluster balances after applying the (unfair) average-linkage algorithm, **(B)** the resultant cluster balances after running Algorithm 11 with parameters $(c, h, k, \varepsilon) = (8, 4, 2, 1/c \cdot \log_2 n)$, **(C)** cluster balances after tuning $c = 4$, **(D)** cluster balances after further tuning $c = 2$. The vertical red line on each plot indicates the balance of the dataset itself.

Part II

Massively Parallel Graph Algorithms

Chapter 10: Streaming and Massively Parallel Algorithms for Edge Coloring

10.1 Introduction

Given a graph $G(V, E)$, an edge coloring of G is an assignment of "colors" to the edges in E such that no two incident edges receive the same color. The goal is to find an edge coloring that uses few colors. Edge coloring is among the most fundamental graph problems and has been studied in various models of computation, especially in distributed and parallel settings.

Denoting the maximum degree in the graph by Δ, it is easy to see that Δ colors are necessary in any proper edge coloring. On the other hand, Vizing's celebrated theorem asserts that $\Delta + 1$ colors are always sufficient [Vizing, 1964]. While determining whether a graph can be Δ colored is NP-hard, a $\Delta + 1$ coloring can be found in polynomial time [Arjomandi, 1982, Gabow et al., 1985]. These algorithms are, however, highly sequential. As a result, in restricted settings, it is standard to consider more relaxed variants of the problem where more colors are allowed [Alon et al., 1986, Barenboim et al., 2016, Fraigniaud et al., 2016, Goldberg et al., 1987, Goldberg and Plotkin, 1987, Harris et al., 2016, Johansson, 1999, Kuhn and Wattenhofer, 2006, Linial, 1992, Luby, 1985, Panconesi and Srinivasan, 1992].

In this paper, we study edge coloring in large-scale graph settings. Specifically, we focus on the *Massively Parallel Computations* (MPC) model and the *Graph Streaming* model.

10.1.1 Massively Parallel Computation

The Model. The MPC model [Beame et al., 2017, Goodrich et al., 2011, Karloff et al., 2010] is a popular abstraction of modern parallel frameworks such as MapReduce, Hadoop, Spark, etc. In this model, there are N machines, each with a space of S words[1] that all run in parallel. The input, which in our case is the edge-set of graph $G(V, E)$, is initially distributed among the machines arbitrarily. Afterwards, the system proceeds in synchronous rounds wherein the machines can perform any arbitrary local computation on their data and can also send messages to other machines. The messages are then delivered at the start of the next round so long as the total messages sent and received by each machine is $O(S)$ for local machine space S. The main parameters of interest are S and the round-complexity of the algorithm, i.e., the number of rounds it takes until the algorithm stops. Furthermore, the total available space over all machines should ideally be linear in the input size, i.e., $S \cdot N = O(|E|)$.

Related Work in MPC. We have seen a plethora of results on graph problems ever since the formalization of MPC. The studied problems include matching and vertex cover [Ahn and Guha, 2015a, Assadi et al., 2019a, Behnezhad et al., 2019e, Czumaj et al., 2018, Ghaffari et al., 2018, Lattanzi et al., 2011, Behnezhad et al., 2018b, Brandt et al., 2018], maximal independent set [Ghaffari et al., 2018, Harvey et al., 2018, Behnezhad et al., 2018b, Brandt et al., 2018], vertex coloring [Chang et al., 2018, Harvey et al., 2018, Parter, 2018, Parter and Su, 2018], as well as graph connectivity and related problems [Andoni et al., 2014, Andoni et al., 2018, Behnezhad et al., 2019d, Jurdzinski and Nowicki, 2018, Bateni et al., 2017]. (This is by no means a complete list of the prior works.)

[1] Throughout the paper, the Stated space bounds are in the number of words that each denotes $O(\log n)$ bits.

We have a good understanding of the complexity of vertex coloring in the MPC model, especially if the local space is near linear in n: Assadi et al. [Assadi et al., 2019a] gave a remarkable algorithm that using $\widetilde{O}(n)$ space per machine, finds a $(\Delta+1)$ vertex coloring in a constant number of rounds. The algorithm is based on a sparsification idea that reduces the number of edges from m to $O(n\log^2 n)$. But this algorithm alone cannot be used for coloring the edges, even if we consider the more relaxed $(2\Delta-1)$ edge coloring problem which is equivalent to $(\Delta+1)$ vertex coloring on the line graph. The reason is that the line-graph has $O(m)$ vertices where here m is the number of edges in the original graph. Therefore even after the sparisification step, we have $\widetilde{O}(m)$ vertices in the graph which is much larger than the local space available in the machines.

Not much work has been done on the edge coloring problem in the MPC model. The only exception is the algorithm of Harvey *et al.* [Harvey et al., 2018] which roughly works by random partitioning the *edges*, and then coloring each partition in a different machine using a sequential $(\Delta+1)$ edge coloring algorithm. The choice of the number of partitions leads to a trade-off between the number of colors used and the space per machine required. The main shortcoming of this idea, however, is that if one desires a $\Delta+\widetilde{O}(\Delta^{1-\Omega(1)})$ edge coloring, then a strongly super linear local space of $n\Delta^{\Omega(1)}$ is required.

Our main MPC result is the following algorithm which uses a more efficient partitioning. The key difference is that we use a *vertex* partitioning as opposed to the algorithm of Harvey *et al.* which partitions the edges.

Result 1 (Theorem 93)**.** *There exists an MPC algorithm that using $\widetilde{O}(n)$ space per machine and $O(m)$ total space, returns a $\Delta+\widetilde{O}(\Delta^{3/4})$ edge coloring in $O(1)$ rounds. This algorithm w.h.p.*

uses $(1+o(1))\Delta$ *colors.*

The algorithm exhibits a tradeoff between the space and the number of colors (see Theorem 93) and can be made more space-efficient as the maximum degree gets larger. For instance, if $\Delta > n^{\epsilon}$ for any constant $\epsilon > 0$, it requires a strictly sublinear space of $n^{1-\Omega(1)}$ to return a $\Delta + o(\Delta)$ edge coloring in $O(1)$ rounds. This is somewhat surprising since all previous non-trivial algorithms in the strictly sublinear regime of MPC require $\omega(1)$ rounds.

Our algorithm can also be implemented in $O(1)$ rounds of *Congested Clique*, leading to a $\Delta + \widetilde{O}(\Delta^{3/4})$ edge coloring there. Prior to our work, no sublogarithmic round Congested Clique algorithm was known even for $(2\Delta - 1)$ edge coloring.

10.1.2 Streaming

The Model. In the standard graph streaming model, the edges of a graph arrive one by one and the algorithm has a space that is much smaller than the total number of edges. A particularly important choice of space is $\widetilde{O}(n)$—which is also known as the *semi-streaming* model [Feigenbaum et al., 2004]—so that the algorithm has enough space to store the vertices but not the edges. For edge coloring, the output is as large as the input, thus, we cannot hope to be able to store the output and report it in bulk at the end. For this, we consider a standard twist on the streaming model where the output is also reported in a streaming fashion. This model is referred to in the literature as the "W-streaming" model [Demetrescu et al., 2006, Glazik et al., 2017]. We particularly focus on one-pass algorithms.

Designing one-pass W-streaming algorithms is particularly challenging since the algorithm cannot "remember" all the choices made so far (e.g., the reported edge colors). Therefore, even

the sequential greedy algorithm for $(2\Delta - 1)$ edge coloring, which iterates over the edges in an arbitrary order an assigns an available to each color upon visiting it, cannot be implemented since we are not aware of the colors used incident to an edge.

Our first result is to show that a natural algorithm w.h.p.[2] provides an $O(\Delta)$ edge coloring if the edges arrive in a random-order.

Result 2 (Theorem 102)**.** *If the edges arrive in a random-order, there is a one-pass $\widetilde{O}(n)$ space W-streaming edge coloring algorithm that always returns a valid edge coloring and w.h.p. uses $(2e + o(1))\Delta \approx 5.44\Delta$ colors.*

If the edges arrive in an arbitrary order, we give another algorithm that requires more colors.

Result 3 (Theorem 103)**.** *For any arbitrary arrival of edges, there is a one-pass $\widetilde{O}(n)$ space W-streaming edge coloring algorithm that succeeds w.h.p. and uses $O(\Delta^2)$ colors.*

These are, to our knowledge, the first streaming algorithms for edge coloring.

10.2 The MPC Algorithm

In this section, we consider the edge coloring problem in the MPC model. Our main result in this section is an algorithm that achieves the following:

Theorem 93. *For any parameter k (possibly dependent on Δ) such that $n/k \gg \log n$, there exists an MPC algorithm with $O(\frac{n\Delta}{k^2} + \frac{n}{k}\sqrt{\Delta \log n/k})$ space per machine and $O(m)$ total space that w.h.p. returns a $\Delta + O(\sqrt{k\Delta \log n})$ edge coloring in $O(1)$ rounds.*

As a corollary of Theorem 93, we get the following algorithm which uses $\widetilde{O}(n)$ space:

[2]Throughout, we use "w.h.p." to abbreviate "with high probability" implying probability at least $1 - 1/\mathsf{poly}(n)$.

Corollary 94. *There exists a randomized MPC algorithm with $\widetilde{O}(n)$ space per machine and $O(m)$ total space that w.h.p. returns a $(1+o(1))\Delta$ edge coloring in $O(1)$ rounds.*

Proof. By setting $k = \Theta(\Delta^{0.5})$ in Theorem 93, we get a $\Delta + O(\Delta^{3/4}\sqrt{\log n})$ edge coloring using $\widetilde{O}(n)$ space. Note that this already is a $(1+o(1))\Delta$ edge coloring if $\Delta = \omega(\log^2 n)$. Otherwise, using a space per machine of $O(n \log^2 n)$, we can simply fit the whole graph into one machine and find a $\Delta+1$ edge coloring there using the known sequential algorithms [Arjomandi, 1982, Gabow et al., 1985]. □

Moreover, assuming that $\Delta = n^{\Omega(1)}$, by setting $k = \Delta^{0.5+\epsilon}$ for a small enough constant $\epsilon \in (0,1)$, we get the following $O(1)$ round algorithm which requires $n^{1-\Omega(1)}$ machine space, which is notably strictly sublinear in n:

Corollary 95. *If $\Delta = n^{\Omega(1)}$, there exists a randomized MPC algorithm with $O(n/\Delta^{2\epsilon}) = n^{1-\Omega(1)}$ space per machine and $O(m)$ total space that w.h.p. returns a $\Delta + O(\Delta^{0.75+\epsilon/2}\sqrt{\log n}) = (1+o(1))\Delta$ edge coloring in $O(1)$ rounds.*

Finally, by setting $k = \sqrt{\Delta} + \log n$, the space required per machine will be $O(n)$. Using a reduction from [Behnezhad et al., 2018a], this leads to an $O(1)$ round Congested Clique $\Delta + \widetilde{O}(\Delta^{3/4})$ edge coloring algorithm.

Corollary 96. *There exists a randomized Congested Clique algorithm that w.h.p. finds a $\Delta + O(\Delta^{3/4}\sqrt{\log n} + \sqrt{\Delta}\log n)$ edge coloring in $O(1)$ rounds.*

We note that the Congested Clique algorithm above is particularly useful, i.e., achieves a $(1+o(1))\Delta$ edge-coloring, for graphs with maximum degree at least $\Delta = \omega(\log^2 n)$. For the lower degree graphs, the additive $O(\sqrt{\Delta}\log n)$ colors exceed Δ and thus may not be negligible.

The Idea Behind the Algorithm. The first step in the algorithm is a random partitioning of the *vertex set* into k groups, $V_1, \ldots, V_k$. We then introduce one subgraph for each vertex subset, called $G_1, \ldots, G_k$, and one subgraph for every pair of groups which we denote as $G_{1,2}, \ldots, G_{1,k}, \ldots, G_{k-1,k}$. Any such G_i is simply the induced subgraph of G on V_i. Moreover, any such $G_{i,j}$ is the subgraph on vertices $V_i \cup V_j$, with edges with one point in V_i and the other in V_j.

The general idea is to assign different *palettes*, i.e., subsets of colors, to different subgraphs so that the palettes assigned to any two neighboring subgraphs (i.e., those that share a vertex) are completely disjoint. A key insight to prevent this from blowing up the number of colors, is that since any two edges from $G_{i,j}$ and $G_{i',j'}$ with $i \neq i'$ and $j \neq j'$ cannot share endpoints by definition, it is safe to use the same color palette for them.

To assign these color palettes, we consider a complete k-vertex graph with each vertex v_i in it corresponding to partition V_i and each edge (v_i, v_j) in it corresponding to the subgraph $G_{i,j}$. We then find a k edge coloring of this complete graph, which exists by Vizing's theorem since maximum degree in it is $k-1$. This edge coloring can actually be constructed extremely efficiently using merely the edges' endpoint IDs. Thereafter, we map each of these k colors to a color palette. By carefully choosing k and the number of colors in each palette, we ensure that: (1) The total number of colors required is close to Δ. (2) Each subgraph $G_{i,j}$ can be properly edge-colored with those colors in its palette. (3) Each subgraph fits the memory of a single machine so that we can put it in whole there and run the sequential edge coloring algorithm on it.

The algorithm outlined above is formalized as Algorithm 12. We start by proving certain bounds on subgraphs' size and degrees.

Algorithm 12 An MPC algorithm for edge coloring.

Input k

Output An edge coloring of a given graph $G = (V, E)$ with maximum degree Δ using $\Psi := \Delta + d\sqrt{k\Delta \log n}$ colors for some large enough constant d.

1: Independently and u.a.r. partition V into k subsets $V_1, \ldots, V_k$.
2: For every $i \in [k]$, let G_i be the induced subgraph of G on V_i.
3: For every $i, j \in [k]$ with $i \neq j$, let $G_{i,j}$ be the subgraph of G including an edge $e \in E$ iff one end-point of e is in V_i and the other is in V_j.
4: Partition $[\Psi]$ into $k+1$ disjoint subsets $C_1, \ldots, C_k, C'$, which we call *color palettes*, in an arbitrarily way such that each palette has exactly $\frac{\Psi}{k+1}$ colors.
5: **for** each graph G_i in parallel **do**
6: Color G_i sequentially in a single machine with palette C'.
7: **end for**
8: ▷ In what follows, we implicitly construct a k edge coloring of a complete k-vertex graph K_k and assign palette C_α to subgraph $G_{i,j}$ where α is the color of edge (i, j) in K_k.
9: **for** each graph $G_{i,j}$ in parallel **do**
10: Color $G_{i,j}$ sequentially in a machine with palette C_α where $\alpha = ((i + j) \mod k) + 1$.
11: **end for**

Claim 97. *W.h.p., every subgraph of type G_i or $G_{i,j}$ has maximum degree $\frac{\Delta}{k} + O(\sqrt{\Delta \log \frac{n}{k}})$ and has at most $O(\frac{n\Delta}{k^2} + \frac{n}{k}\sqrt{\Delta \log n/k})$ edges.*

Proof. Let us start with bounding the degree of an arbitrary vertex $v \in V_i$ in subgraph G_i. The degree of vertex v in G_i is precisely the number of its neighbors that are assigned to partition V_i. Since there are k partitions, the expected degree of v in G_i is $\deg_G(v)/k \leq \Delta/k$. Furthermore, since the assignment of vertices to the partitions is done independently and uniformly at random, by a simple application of Chernoff bound, v's degree in G_i should be highly concentrated around its mean. Namely, with probability at least $1 - n^{-2}$, it holds that $\deg_{G_i}(v) \leq \frac{\Delta}{k} + O(\sqrt{\Delta \log n/k})$. Now, a union bound over the n vertices in the graph, proves that the degree of all vertices in their partitions should be at most $\frac{\Delta}{k} + O(\sqrt{\Delta \log n/k})$ with probability $1 - 1/n$.

Bounding vertex degrees in subgraphs of type $G_{i,j}$ also follows from essentially the same argument. The only difference is that we have to union bound over $n \cdot k$ choices, as we would like to bound the degree of any vertex v with say $v \in V_i$ in k subgraphs $G_{i,1}, \ldots, G_{i,k}$. Nonetheless,

since $k \leq n$, there are still poly(n) many choices to union bound over. Thus, by changing the constants in the lower terms of the concentration bound, we can achieve the same high probability result.

Finally, we focus on the number of edges in each of the subgraphs. Each partition V_i has n/k vertices in expectation since the n vertices are partitioned into k groups independently and uniformly at random. A simple application of Chernoff and union bounds, implies that the number of vertices in each partition V_i is at most $O(\frac{n}{k})$ w.h.p., so long as $n/k \gg \log n$, which is the case. Since the number of edges in each partition is less than the number of vertices times max degree, combined with the aforementioned bounds on the max degree, we can bound the number of edges in G_i and $G_{i,j}$ for any i and j by

$$O\left(\frac{n}{k}\right) \cdot O\left(\frac{\Delta}{k} + \sqrt{\frac{\Delta}{k}\log n}\right) = O\left(\frac{n\Delta}{k^2} + \frac{n}{k}\sqrt{\frac{\Delta}{k}\log n}\right),$$

which is the claimed bound. □

Next, observe that we use palettes $C_1, \ldots, C_{k+1}, C'$, each of size $\frac{\Psi}{k+1}$ to color the subgraphs. We need to argue that the maximum degree in each subgraph is at most $\frac{\Psi}{k+1} - 1$ to be able to argue that using Vizing's theorem in one machine, we can color any of the subgraphs with the assigned palettes. This can indeed be easily guaranteed if the constant d is large enough:

Observation 98. *If constant d in Algorithm 12 is large enough, then maximum degree of every graph is at most $\frac{\Psi}{k+1} - 1$, w.h.p.*

Proof. We have $\Psi = \Delta + d\sqrt{k\Delta \log n}$ in Algorithm 12, therefore:

$$\frac{\Psi}{k+1} = \frac{\Delta}{k+1} + \frac{d\sqrt{k\Delta \log n}}{k+1} = \frac{\Delta}{k} + \Theta(\sqrt{\Delta \log n/k}),$$

where the hidden constants in the second term of the last equation can be made arbitrarily large depending on the choice of constant d. On the other hand, recall from Claim 97 that the maximum degree in any of the subgraphs is also at most $\frac{\Delta}{k} + O(\sqrt{\Delta \log n/k})$. Thus, the palette sizes are sufficient to color the subgraphs if d is a large enough constant. □

We are now ready to prove the algorithm's correctness.

Lemma 99. *Algorithm 12 returns a proper edge coloring of G using $\Delta + O(\sqrt{k\Delta \log n})$ colors.*

Proof. The algorithm clearly uses $\Psi = \Delta + O(\sqrt{k\Delta \log n})$ colors, it remains to argue that the returned edge coloring is proper. Each subgraph (of type G_i or $G_{i,j}$) is sent to a single machine and edge-colored there using the palette that it is assigned to. Since by Observation 98, each palette has at least $\Delta' + 1$ colors for Δ' being the max degree in the subgraphs, there will be no conflicts in the colors associated to the edges within a partition. We only need to argue that two edges e and f sharing a vertex v that belong to two different subgraphs are not assigned the same color. Note that all subgraphs of type G_i are vertex disjoint and all receive the special color palette C', thus there cannot be any conflict there. To complete the proof, it suffices to prove that any two subgraphs $G_{i,j}$ and $G_{i',j'}$ that share a vertex receive different palettes. Note that in this case, either $i = i'$ or $j = j'$ by the partitioning. Assume w.l.o.g. that $i = i'$ and thus $j \neq j'$. Based on Algorithm 12 for $G_{i,j}$ and $G_{i',j'}$ to be assigned the same color palette, it should hold

that

$$((i+j) \bmod k) + 1 = ((i'+j') \bmod k) + 1.$$

Since $i = i'$, this would imply that $(j \bmod k) = (j' \bmod k)$, though this would not be possible given that both j and j' are in $[k]$ and that $j \neq j'$. Therefore, any two subgraphs that share a vertex receive different palettes and thus there cannot be any conflicts, completing the proof. □

Next, we turn to prove the space bounds.

Lemma 100 (Implementation and Space Complexity)**.** *Algorithm 12 can be implemented with total space $O(m)$ and space per machine of $O(\frac{n\Delta}{k^2} + \frac{n}{k}\sqrt{\Delta \log n/k})$ w.h.p.*

Proof. We start with an implementation that uses the specified space per machine but can be wasteful in terms of the total space, then describe how we can overcome this problem and also achieve an optimal total space of $O(m)$.

We can use $k + \binom{k}{2}$ machines, each with a space of size $O(\frac{n\Delta}{k^2} + \frac{n}{k}\sqrt{\Delta \log n/k})$ to assign colors to the edges in parallel. The first $m_1, \ldots, m_k$ machines will be used for edge coloring on $G_1, G_2, \ldots, G_k$ respectively. The other $m_{k+1}, \ldots, m_{k+\binom{k}{2}}$ machines will be used for edge coloring on the $G_{i,j}$ graphs. Lemma 97 already guarantees that each subgraph has size $O(\frac{n\Delta}{k^2} + \frac{n}{k}\sqrt{\Delta \log n/k})$ w.h.p., and thus fits the memory of a single machine.

In the implementation discussed above, since the machines use $\widetilde{O}(n\Delta/k^2)$ space and there are $O(k^2)$ machines, the total memory can be $\widetilde{O}(n\Delta)$ which may be much larger than $O(m)$. This is because we allocate $O(n\Delta/k^2)$ space to each machine regardless of how much data it actually received. Though, observe that each edge of the graph belongs to exactly one of the subgraphs, i.e., the machines together only handle a total of $O(m)$ data. So we must consolidate into fewer machines. We do this by putting multiple subgraphs in each machine.

We start by recalling a sorting primitive in the MPC model which was proved in [Goodrich et al., 2011]. Basically, if there are N items to be sorted and the space per machine is $N^{\Omega(1)}$, then the algorithm of [Goodrich et al., 2011] sorts these items into the machines within $O(1)$ rounds. To use this primitive, we first label each edge $e = (u, v)$ of the graph by its subgraph name (e.g. G_i or $G_{i,j}$) which can be determined solely based on the end-points of the edge. After that, we sort the edges based on these labels. This way, all the edges inside each subgraph can be sent to the same machine within $O(1)$ rounds while also ensuring that the total required space remains $O(m)$. □

The algorithm for Theorem 93 was formalized as Algorithm 12. We showed in Lemma 99 that the algorithm correctly finds an edge coloring of the graph with the claimed number of colors. We also showed in Lemma 100 that the algorithm can be implemented with $O(m)$ total space and $O(\frac{n\Delta}{k^2} + \frac{n}{k}\sqrt{\Delta \log n/k})$ space per machine. This completes the proof of Theorem 93.

10.3 Streaming Algorithms

We start in Section 10.3.1 by describing our streaming algorithm and its analysis when the arrival order is random. Then in Section 10.3.2, we give another algorithm for adversarial order streams.

10.3.1 Random Edge Arrival Setting

In this section, we give a streaming algorithm for $O(\Delta)$ edge coloring using $\widetilde{O}(n)$ space where the edges come in a random stream. That is, a permutation over the edges is chosen uniformly at random and then the edges arrive according to this permutation.

We first note that if $\Delta = O(\log n)$ then the problem is trivial as we can store the whole graph and then report a $\Delta+1$ edge coloring (even without knowledge of Δ). As such, we assume $\Delta = \omega(\log n)$.

The algorithm — formalized as Algorithm 13 — maintains a counter c_v for each vertex v. At any point during the algorithm, this counter c_v basically denotes the highest color number used for the edges incident to v so far, plus 1. Therefore, upon arrival of an edge (u, v), it is safe to color this edge with $\max(c_u, c_v)$ as all edges incident to u and v have a color that is strictly smaller than this. Then, we increase the counters of both v and u to $\max(c_u, c_v) + 1$. It is not hard to see that the solution is always a valid coloring, in the remainder of this section, we mainly focus on the number of colors required by this algorithm and show that w.h.p., it is only $O(\Delta)$ for random arrivals.

Algorithm 13 Edge coloring for random streams.

Output A feasible coloring $\mathcal{C} : E \to [\Psi]$ for a given graph $G = (V, E)$ with maximum degree Δ in a random stream

1: $c_v \leftarrow 0 \quad \forall v \in V$
2: **while** (u, v) is read from stream **do**
3: $\quad \mathcal{C}(u, v) \leftarrow \max(c_u, c_v)$
4: $\quad c_u, c_v \leftarrow \mathcal{C}(u, v) + 1$
5: **end while**

We start by noting that this algorithm can actually be extremely bad if the order is adversarial. To see this, consider a path of size n. In an adversarial stream where the edges arrive in the order of the path, Algorithm 13 uses as many as $n - 1$ colors while the maximum degree is only 2! It is easy to see why this example is very unlikely to occur in random order streams: For a fixed path, it is very unlikely that the edges are randomly ordered in this very specific way.

To make this intuition rigorous for general graphs, we first prove the following crucial lemma which gives us the correct parameter to bound.

Lemma 101. *Let Ψ be the size of the longest monotone (in the order of arrival) path in the line-graph of G. Then Algorithm 13 uses exactly Ψ colors.*

Proof. Take a monotone path $v_1, v_2, \ldots, v_\Psi$ in the line-graph of G and let $e_1, e_2, \ldots, e_\Psi$ be the edges of the original graph that correspond to these vertices respectively, i.e., e_1 arrives before e_2 which arrives before e_3 and so on. Since for any i, v_i and v_{i+1} are neighbors in the line-graph, then e_i and e_{i+1} should share an end-point v. This means that at the time of arrival of e_{i+1}, we have $c_v \geq \mathcal{C}(e_i) + 1$ which in turn, implies $\mathcal{C}(e_\Psi) > \mathcal{C}(e_{\Psi-1}) > \ldots > \mathcal{C}(e_1)$. Therefore, $\mathcal{C}(e_\Psi) \geq \Psi$.

On the other hand, suppose that there is an edge $e_1 = (u, v)$ for which $\mathcal{C}(e_1) = \Psi$ in Algorithm 13. This means that at least one of c_u or c_v equals Ψ when e_1 arrives, say c_u w.l.o.g. Let e_2 be the last edge incident to u that has arrived before e_1. It should hold that $\mathcal{C}(e_2) = \Psi - 1$. Using the same argument, for each $1 < i \leq \Psi$, we can find a neighboring edge e_i such that $\mathcal{C}(e_i) = \mathcal{C}(e_{i-1}) - 1$. This way, we end up with a sequence $e_1, \ldots, e_\Psi$ of edges, the path corresponding to this sequence in the line graph will be a monotone path of length Ψ, completing the proof. □

Theorem 102. *There is a streaming edge coloring algorithm that for any graph $G = (V, E)$ uses at most $(2e + \epsilon)\Delta \approx 5.44\Delta$ colors w.h.p. for any constant $\epsilon > 0$ given that the edges in E arrive in a random order.*

Proof. We first prove that Algorithm 13 gives us a feasible coloring of graph G. Consider two edges $e_1 = (u, v)$ and $e_2 = (u, v')$ incident to vertex u such that e_1 appears earlier than e_2 in the stream. For any edge e we represent by $\mathcal{C}(e)$ the color assigned to that by the algorithm. After the algorithm colors e_1 with $\mathcal{C}(e_1)$, it sets c_u to $\mathcal{C}(e_1) + 1$. Thus, c_u is at least $\mathcal{C}(e_1) + 1$ when e_2 arrives and $\mathcal{C}(e_2) \geq \mathcal{C}(e_1) + 1$ consequently. Therefore, $\mathcal{C}(e_2) > \mathcal{C}(e_1)$ for any pair of edges

incident to a common vertex, and $\mathcal{C}$ is a feasible coloring.

Next, for some constant α that we fix later, we show that the probability that an edge is assigned a color number at least $\alpha\Delta$ is at most n^{-c} for some constant $c \geq 2$, implying via a union bound over all the edges that indeed w.h.p., $\Psi \leq \alpha\Delta$.

We showed in Lemma 101 that if the number of colors Ψ used is $\alpha\Delta$, then there should exist a monotone path in the line-graph with size at least $\alpha\Delta$. Let $e_0, e_2, \ldots, e_{\alpha\Delta}$ be the corresponding edges to this path. Thus, it suffices to bound the probability of this event. Let Π denote the set of all such paths in the line graph. For a specific path $\pi \in \Pi$, the probability that it is monotone is $1/(\alpha\Delta)!$. Call this event X_π. On the other hand, we can upper bound the number of such paths by $(2\Delta)^{\alpha\Delta}$, i.e., $|\Pi| \leq (2\Delta)^{\alpha\Delta}$. This follows from the fact that each path should start from the corresponding vertex to e_0 in the line-graph, and that maximum degree in the line graph is $2\Delta - 2$ (which is the upper bound on the number of neighboring edges to each edge). Thus:

$$\Pr[\mathcal{C}(e_0) \geq \alpha\Delta] = \Pr[\bigvee_{\pi \in \Pi} X_\pi] \leq \sum_{\pi \in \Pi} \Pr[X_\pi = 1] \leq \frac{(2\Delta)^{\alpha\Delta}}{(\alpha\Delta)!},$$

where the last inequality is obtained by replacing $\Pr[X_\pi = 1]$ and $|\Pi|$ by the aforementioned bounds. Taking the logarithm of each side of the inequality, we get

$$\begin{aligned}
\ln(\Pr[\mathcal{C}(e_0) \geq \alpha\Delta]) &\leq \alpha\Delta \ln(2\Delta) - \ln((\alpha\Delta)!) \\
&\leq \alpha\Delta \ln(2\Delta) - ((\alpha\Delta + 1/2)\ln(\alpha\Delta) - \alpha\Delta) && (10.1) \\
&= \alpha\Delta \ln(2e/\alpha) - 1/2 \ln(\alpha\Delta) && (10.2) \\
&\leq \alpha\Delta \ln(2e/\alpha). && (10.3)
\end{aligned}$$

To obtain (10.1), we use Stirling's approximation of factorials to lower-bound $\ln((\alpha\Delta)!)$. Finally, we rearranged terms to imply (10.2). By plugging in $\alpha = 2e(1+\epsilon)$, we get

$$\begin{aligned}
\ln(\Pr[\mathcal{C}(e_0) \geq 2e(1+\epsilon)\Delta]) &\leq 2e(1+\epsilon)\Delta \ln\left(\frac{1}{1+\epsilon}\right) \\
&= -2e(1+\epsilon)\ln(1+\epsilon)\Delta \\
&\leq -2e(1+\epsilon)\ln(1+\epsilon)\frac{c}{2e(1+\epsilon)\ln(1+\epsilon)}\ln(n) \qquad (10.4) \\
&= -c\ln(n)
\end{aligned}$$

Since $\Delta = \omega(\log(n))$, we have $\Delta > c'\ln(n)$ for any constant c'. Inequality (10.4) follows from setting $c' = c/(2e(1+\epsilon)\ln(1+\epsilon))$ in $\Delta > c'\ln(n)$, where c is the constant for which we want to show the probability is upper-bounded by n^{-c}. Hence,

$$\Pr[\mathcal{C}(e_0) \geq 2e(1+\epsilon)\Delta] \leq n^{-c}.$$

Thus, Algorithm 13 returns a feasible coloring of the input graph G using at most $2e(1+\epsilon)\Delta$ colors, for any constant $\epsilon > 0$ w.h.p. if the edges arrive in a random order. □

To further evaluate the performance of Algorithm 13, we implemented and ran it for cliques of different size. The result of this experiment is provided in Table 10.1. The numbers are obtained by running the experiment 100 times and taking the average number of colors used. As it can be observed from Table 10.1, for cliques of size 100 to 1000, the number of colors used by the algorithm is in range $[3.3\Delta, 3.9\Delta]$ and it slightly increases by the size of the graph. Our analysis, however, shows that it should never exceed 5.44Δ.

Clique Size	100	200	300	400	500	600	700	800	900	1000
Colors Used	3.363Δ	3.563Δ	3.665Δ	3.717Δ	3.756Δ	3.787Δ	3.815Δ	3.838Δ	3.849Δ	3.863Δ

Table 10.1: The number of colors used by Algorithm 13 on cliques averaged over 100 trials.

10.3.2 Adversarial Edge Arrival Setting

In this section, we turn to arbitrary (i.e., adversarial) arrivals of the edges. We assume that the adversary is *oblivious*, i.e., the order of the edges is determined before the algorithm starts to operate so that the adversary cannot abuse the random bits used by the algorithm. Having this assumption, we give a randomized algorithm that w.h.p., outputs a valid edge coloring of the graph using $O(\Delta^2)$ colors while using $\tilde{O}(n)$ space. The algorithm is formalized as Algorithm 14. We note that this algorithm, as Stated, requires knowledge of Δ. However we later show that we can get rid of this assumption. Overall, we get the following result:

Theorem 103. *Given a graph G with maximum degree Δ, there exists a one pass streaming algorithm, that outputs a valid edge coloring of the G using $O(\Delta^2)$ colors w.h.p., using $\widetilde{O}(n)$ memory.*

Consider two vertices v and u and their string of random bits r_v and r_u defined in the algorihtm. Let $d_{u,v}$ be the smallest index i where $r_{u,i} \neq r_{v,i}$. Upon arrival of an edge $e = (u, v)$, we first find $i := d_{u,v}$. If $\Delta 2^{-i} > \log n$, we color the edge immediately. Otherwise, we store it. We will show that all the stored edges fit in the memory thus after reading all the stream we can color them with a palette of at most $\Delta + 1$ new colors. In the algorithm, for any vertex v and any $i \in [\log n]$, we define a counter $c_{u,i}$. If $\Delta 2^{-i} > \log n$ for any edge e, then we immediately assign e a color which is represented by a tuple $(c_{u,i}, c_{v,i}, i)$. Then, we increase counters $c_{u,i}$ and $c_{v,i}$. Note that we say two colors are the same if all three elements of them are equal. We first show

Algorithm 14 Edge coloring in the adversarial order

Output A feasible coloring for a given graph $G = (V, E)$ with maximum degree Δ

for any vertex $v \in V$ **do**
  $r_v \leftarrow$ a sequence of $\log(n)$ independent random bits
  for any $i \in [\log n]$ **do**
    $c_{v,i} \leftarrow 0$
  end for
end for
for any edge $e = (u, v)$ in the stream **do**
  Let i be the smallest index for which $r_{v,i} \neq r_{u,i}$.
  if $\Delta 2^{-i} > \log n$ **then**
    if $r_{u,i} = 1$ **then**
      Assign color $(c_{u,i}, c_{v,i}, i)$ to e.
    else
      Assign color $(c_{v,i}, c_{u,i}, i)$ to e.
    end if
    Increase both $c_{v,i}$ and $c_{u,i}$ by one.
  else
    Store edge e.
  end if
end for
Color the stored edges using a new set of colors.

that this gives us a valid coloring, which means it does not assign the same color to two edges adjacent to the same vertex. We use proof by contradiction. Assume that our algorithm assigns the same color to edges $e_1 = (u, v_1)$ and $e_2 = (u, v_2)$ adjacent to vertex u. None of them can be from the stored edges since we color them using a new palette. This means that $d_{u,v_1} = d_{u,v_2}$. Let us denote it by i. Without loss of generality, we assume that $r_{u,i} = 1$ and that in the input stream e_1 arrives before e_2. Note that the first element of the colors (which are tuples) assigned to these edges is the value of counter $c_{u,i}$ when they arrive. However, the algorithm increases $c_{u,i}$ by one after arrival of e_1 thus the colors assigned to e_1 and e_2 cannot be the same.

Now, it suffices to show that the total number of colors used by the algorithm is $O(\Delta^2)$. Given a vertex v, and a number $l \in [\log n]$ let us compute an upper-bound for counter $c_{v,i}$. Let N_v be the set of neighbors of this vertex and let $N_{v,i}$ be the set of neighbors like u where $d_{v,u} = i$.

We know that $c_{v,i} = |N_{v,i}|$, thus given any vertex v and $i \in [\log(n)]$, we need to find a bound for $|N_{v,i}|$. Given any edge $e = (v, u)$ the probability of e being in set $N_{v,i}$ is 2^{-i} which means $\mathbb{E}[|N_{v,i}|] = \deg(v)2^{-i}$ where $\deg(v)$ is the degree of vertex v in the input graph.

Using a simple application of the Chernoff bound, for any vertex v, we get:

$$Pr\Big[|N_{v,i}| \geq \deg(v)2^{-i} + O\big(\sqrt{\deg(v)2^{-i}\log n}\big)\Big] \leq \frac{1}{n^c}.$$

Setting c to be a large enough constant, one can use union bound and show that w.h.p., for any vertex v and $i \in [\log n]$ where $\deg(v)2^{-i} \geq \log n$, we have $|N_{v,i}| \leq O(\deg(v)2^{-i})$.

Having this, we conclude that for any $i \in [\log n]$, where $\Delta 2^{-i} > \log n$, the number of colors used by the algorithm whose third element is i is at most $O(\Delta^2 2^{-2i})$ since the first and the second element of the color can get at most $O(\Delta 2^{-i})$ different values. Therefore, the total number of colors used for any such i is at most $O\big(\sum_{i\in[\log n]} \Delta^2 2^{-2i}\big) = O(\Delta^2)$. We will also show that the stored edges fit in the memory and thus we can color them using $O(\Delta)$ new colors. As a result the total number of colors used is $O(\Delta^2)$.

To give an upper-bound for the number of stored edges we first show that the expected number of stored edges for each vertex is $O(\log n)$. Let $j := \log(\frac{\Delta}{\log n})$. Recall that we store an edge (u, v) when $\Delta 2^{-d_{u,v}} < \log n$. Thus the expected number of stored edges adjacent to a single vertex v is at most

$$\sum_{j\leq i\leq \log n} d_v 2^{-i} \leq \sum_{j\leq i\leq \log n} \Delta 2^{-i} \leq \sum_{j\leq i\leq \log n} \log(n) 2^{-i+j} = O(\log n).$$

To get the last equation we use the fact that $\Delta 2^{-j} \leq \log n$. By a similar argument that we

used above (using Chernoff and Union bounds), with a high probability the total number of stored edges is $O(n \log n)$ which can be stored in the memory. Therefore the proof of this theorem is completed.

Knwoledge of Δ. As written, our algorithm depends on the knowledge of Δ because we must check $\Delta 2^{-i} > \log n$. We can get rid of this condition by keeping track of the degree $\deg_v^H$ of a vertex in the subgraph H we have seen so far, and then computing the max degree $\deg_{max}^H$. This only requires an additional $O(n)$ space. Thereafter, instead of checking if $\Delta 2^{-i} > \log n$, we check if $\deg_{max}^H 2^{-i} > \log n$. Whenever $\deg_{max}^H$ increases, we iterate over all stored edges and recompute whether or not $\deg_{max}^H 2^{-i} > \log n$. If so, we color the edge and remove it from the buffer, else we keep it. It is easy to see that this will not exceed the space bounds because at any timestep, we can assume the input graph was H in the first place. Then its max degree is $\Delta_H = \deg_{max}^H$, and we can apply the same argument for the space bounds as before, but using Δ_H instead of Δ. All other parts of the proof still hold. Therefore our algorithm does not require knowledge of Δ.

Finally, we remark that if one allows more space, then one can modify Algorithm 14 to use fewer number of colors. Though we focused only on the $\widetilde{O}(n)$ memory regime.

10.4 Open Problems

We believe the most notable future direction is to improve the number of colors used in our streaming algorithms. Specifically, our streaming algorithm for adversarial arrivals requires $O(\Delta^2)$ colors. A major open question is whether this can be improved to $O(\Delta)$ while also keeping the memory near-linear in n. Also for random arrival streams, we showed that Algorithm 13

achieves a 5.44Δ coloring and showed, experimentally, that it uses at least 3.86Δ colors. A particularly interesting open question is whether there is an algorithm that uses arbitrarily close to 2Δ colors using $\widetilde{O}(n)$ space in random arrival streams.

Chapter 11: Matching Affinity Clustering: Improved Hierarchical Clustering at Scale with Guarantees

11.1 Introduction

Clustering is one of the most prominent methods to provide structure, in this case clusters, to unlabeled data. It requires a single parameter k for the number of clusters. Hierarchical clustering elaborates on this structure by adding a hierarchy of clusters contained within superclusters. This problem is unparameterized, and takes in data as a graph whose edge weights represent the similarity or dissimilarity between data points. A hierarchical clustering algorithm outputs a tree T, whose leaves represent the input data, internal nodes represent the merging of data and clusters into clusters and superclusters, and root represents the cluster of all data.

Obviously it is more computationally intensive to find T as opposed to a flat clustering. However, having access to such a structure provides two main advantages: (1) it allows a user to observe the data at different levels of granularity, effectively querying the structure for clusterings of size k without recomputation, and (2) it constructs a history of data relationships that can yield additional perspectives. The latter is most readily applied to phylogenetics, where dendrograms depict the evolutionary history of genes and species [Kraskov et al., 2003]. Hierarchical clustering in general has been used in a number of other unsupervised applications. In this paper, we

explore four important qualities of a strong and efficient hierarchical clustering algorithm:

1. **Theoretical guarantees**. Previously, analysis of hierarchical clustering algorithms has relied in experimental evaluation. While this is one indicator for success, it cannot assure performance across a large range of datasets. Researchers combat this by considering optimization functions to evaluate broader guarantees [Charikar et al., 2004, Lin et al., 2006]. One function that has received significant attention recently [Charikar et al., 2019b, Cohen-Addad et al., 2018] is a hierarchical clustering cost function proposed by [Dasgupta, 2016]. This function is simple and intuitive, however, [Charikar and Chatziafratis, 2017] showed that it is likely not constant-factor approximable. To overcome this, we examine its dual, revenue, proposed by [Moseley and Wang, 2017], which considers a graph with *similarity*-based edge weights. For *dissimilarity*-based edge weights, we look to [Cohen-Addad et al., 2018]'s value, another cost-inspired function. We are interested in constant factor approximations for these functions.

2. **Empirical performance**. As theoretical guarantees are often only intuitive proxies for broader evaluation, it is still important to evaluate the empirical performance of algorithms on specific, real datasets. Currently, [Bateni et al., 2017]'s Affinity Clustering remains the state-of-the-art for scalable hierarchical clustering algorithms with strong empirical results. With Affinity Clustering as an inspirational baseline for our algorithm, we strive to preserve and, hopefully, extend Affinity Clustering's empirical success.

3. **Balance**. One downside of algorithms like Affinity Clustering is that they are prone to creating extremely unbalanced clusters. There are a number of natural clustering problems where balanced clusters are preferable or more accurate for the problem, for example, clus-

tering a population into genders. Some more specific applications include image collection clustering, where balanced clusters can make the database more easily navigable [Dengel et al., 2011], and wireless sensor networks, where balancing clusters of sensor nodes ensures no cluster head gets overloaded [Amgoth and Jana, 2014]. Here, we define balance as the minimum ratio between cluster sizes.

4. **Scalability**. Most current approximations for revenue are serial and do not ensure performance at scale. We achieve scalability through distributed computation. Clustering itself, as well as many other big data problems, has been a topic of interest in the distributed community in recent years [Chitnis et al., 2015, Chitnis et al., 2016, Ghaffari et al., 2019b]. In particular, hierarchical clustering has been studied by [Jin et al., 2013, Jin et al., 2015], but only [Bateni et al., 2017] has attempted to ensure theoretical guarantees through the introduction of a Steiner-based cost metric. However, they provide little motivation for its use. Therefore, we are interested in evaluating distributed algorithms with respect to more well-founded optimization functions like revenue and value.

   For our distributed model, we look to *Massively Parallel Communication* (MPC), which was used to design Affinity Clustering. MPC is a restrictive, theoretical abstraction of *MapReduce*: a popular programming framework famous for its ease of use, fault tolerance, and scalability [Dean and Ghemawat, 2008]. In the MPC model, individual machines carry only a fraction of the data and execute individual computations in rounds. At the end of each round, machines send limited messages to each other. Complexities of interest are the number of rounds and the individual machine space. This framework has been used in the analysis for many large-scale problems in big data, including clustering [Im et al.,

2017, Ludwig, 2015, Ghaffari et al., 2019b]. It is a natural selection for this work.

11.1.1 Related Work

There exist algorithms that can achieve up to two of these qualities at a time. Affinity Clustering, notably, exhibits good empirical performance and scalability using MPC. While [Bateni et al., 2017] describe some minor theoretical guarantees for Affinity Clustering, we believe that proving an algorithm's ability to optimize for revenue and value is a stronger and more well-founded result due to their popularity and relation to Dasgupta's cost function. A simple random divisive algorithm proposed by [Charikar and Chatziafratis, 2017] was shown to achieve a $1/3$ expected approximation for revenue and can be efficiently implemented using MPC. However, it is notably nondeterministic, and we show that it does not exhibit good empirical performance. Similarly, balanced partitioning may achieve balanced clusters, but it is unclear whether it is scalable, and it has not been shown to achieve strong theoretical guarantees.

For both revenue and value, Average Linkage achieves near-state-of-the-art $1/3$ and $2/3$-approximations respectively [Moseley and Wang, 2017, Cohen-Addad et al., 2018]. [Charikar et al., 2019a] marginally improves these factors to $1/3 + \epsilon$ for revenue and $2/3 + \epsilon$ for value, through semi-definite programming (SDP, a non-distributable method). However, since value and revenue both strove to characterize Average Linkage's optimization goal, and this was only marginally beat by an SDP, we do not expect to surpass Average Linkage in the restrictive distributed context.

11.1.2 Our contributions

In this work, we propose a new algorithm, *Matching Affinity Clustering*, for distributed hierarchical clustering. Inspired by Affinity Clustering's reliance on the minimum spanning tree in order to greedily merge clusters [Bateni et al., 2017], Matching Affinity Clustering merges clusters based on iterative matchings. It notably generalizes to both the edge weight similarity and dissimilarity contexts, and achieves all four desired qualities.

In Section 11.4, we **theoretically motivate** Matching Affinity Clustering by proving it achieves a good approximation for both revenue and value (the latter depending on the existence of an MPC minimum matching algorithm), nearing the bounds achieved by Average Linkage:

Theorem 104. *In the revenue context (where edge weights are data similarity), with $\widetilde{O}(n)$ machine space, Matching Affinity Clustering achieves:*

- *a $(1/3-\epsilon)$-approximation for revenue in $O(\log(n)\log\log(n)\cdot(1/\epsilon)^{O(1/\epsilon)})$ rounds when $n=2^N$,*
- *and a $(1/9-\epsilon)$-approximation for revenue in $O(\log(nW)\log\log(n)\cdot(1/\epsilon)^{O(1/\epsilon)})$ rounds in general.*

Theorem 105. *Assume there exists an MPC algorithm that achieves an α-approximation for minimum weight k-sized matching in $O(f(n))$ rounds and $\widetilde{O}(n)$ machine space. In the value context (where edge weights are data distances) and in $O(f(n)\log(n))$ rounds with $\widetilde{O}(n)$ machine space, Matching Affinity Clustering achieves:*

- *a $\frac{2}{3}\alpha$-approximation for value when $n=2^N$,*

- *and a $\frac{1}{3}\alpha$-approximation for value in general.*

Furthermore, in Theorem 106, we prove that Matching Affinity Clustering can give no guarantees with respect to revenue or value. The discussion and proof of this theorem can be found in the Appendix.

Theorem 106. *Affinity Clustering cannot achieve better than a $O(1/n)$-factor approximation for revenue or value.*

We also present an efficient and near-optimal MPC algorithm for k-sized maximum matching in Theorem 112 in Section 11.3. This is used by Matching Affinity Clustering.

To evaluate the **empirical performance** of our algorithm, we run [Bateni et al., 2017]'s experiments used for Affinity Clustering on small-scale datasets in Section 11.5. We find Matching Affinity Clustering performs competitively with respect to state-of-the-art algorithms. On filtered, balanced data, we find that Matching Affinity Clustering consistently outperforms other algorithms by at least a small but clear margin. This implies Matching Affinity Clustering may be more useful on balanced datasets than Affinity Clutsering.

To confirm the **balance** of our algorithm, we are able to prove that Matching Affinity Clustering achieves perfectly balanced clusters on datasets of size 2^N, and otherwise guarantee near balance (a cluster size ratio of at most 2). See Lemma 113. This was also confirmed in our empirical evaluation in Section 11.5.

Finally, we show in Section 11.4 that Matching Affinity Clustering is highly **scalable** because it was designed in the same MPC framework as Affinity Clustering. We provide similar complexity guarantees to Affinity Clustering.

Matching Affinity Clustering is ultimately a nice, simply motivated successor to Affinity

Clustering that achieves all four desired qualities: empirical performance, theoretical guarantees, balance, and scalability. No other algorithm that we know of does this.

11.2 Background

In this section, we describe basic notation, hierarchical clustering cost functions, and Massively Parallel Communication (MPC).

11.2.1 Preliminaries

The standard hierarchical clustering problem takes in a set of data represented as a graph where weights on edges measure similarity or dissimilarity between data. In this paper, edge weights, denoted $w_G(u, v)$ for a graph G and may be similarities or differences as specified.

11.2.2 Optimization functions

Consider some hierarchical tree, T. We say $i \vee j$ for leaves i and j is the least common ancestor of i and j. The subtree rooted at an interior vertex v is $T[v]$, therefore the subtree representing the smallest cluster that contains both i and j is $T[i \vee j]$. Let $\text{leaves}(T[v])$ be the set of leaves in $T[v]$, and $\text{non-leaves}(T[v])$ be the set of all of the leaves of T but not $T[v]$. Now we can describe Dasgupta's function.

Definition 107 ([Dasgupta, 2016]). *Dasgupta's **cost** function of tree T on graph G with similarity-based edge weights w_G is a minimization function.*

$$\text{cost}_G(T) = \sum_{i,j \in V(G)} w_G(i, j) |\text{leaves}(T[i \vee j])|.$$

To minimize edge weight contribution, we want a small $|\operatorname{leaves}(T[i \vee j])|$ for heavy edges. This ensures that heavy edges will be merged earlier in the tree. To calculate this, it is easier to break it down into a series of merge costs for each node in T. It counts the costs that accrue due to the merge at that node so that we can keep track of the cost throughout the construction of T. It is defined as:

Definition 108 ([Moseley and Wang, 2017]). *The **merge cost** of a node in T which merges disjoint clusters A and B is:*

$$\begin{aligned}\operatorname{mergecost}_G(A,B) =& |B| \sum_{a\in A, c\in G\setminus(A\cup B)} w_G(a,c) \\ &+ |A| \sum_{b\in B, c\in G\setminus(A\cup B)} w_G(b,c).\end{aligned}$$

This breaks down the cost of a hierarchy tree into a series of merge costs. Consider some edge, (i,j). At each merge containing exclusively i or j, this edge contributes $w_G(i,j)$ times the size of the other cluster. In the hierarchical tree, this counts how many vertices accrue during merges along the paths from i and j to $i \vee j$. However, this does not account for the leaves i or j themselves, so we need to add $w_G(i,j)$ two extra times in addition to each merge. This means we can derive the total cost from the merge costs as: $\operatorname{cost}_G(T) = 2\sum_{i,j\in V(G)} w_G(i,j) + \sum_{\text{merges } A,B} \operatorname{mergecost}_G(A,B)$.

Next, we consider Moseley and Wang's dual to Dasgupta's function [Moseley and Wang, 2017].

Definition 109 ([Moseley and Wang, 2017]). *The **revenue** of tree T on graph G with similarity-*

based edge weights is a maximization function.

$$\text{rev}_G(T) = \sum_{i,j \in V(G)} w_G(i,j) |\text{ non-leaves}(T[i \vee j])|.$$

We can, in a similar fashion to Dasgupta's cost function, break revenue down into a series of merge revenues.

Definition 110. *[Moseley and Wang, 2017]. The **merge revenue** of a node in T which merges disjoint clusters A and B is:*

$$\text{mergerev}_G(A,B) = (n - |A| - |B|) \sum_{a \in A, b \in B} w_G(a,b).$$

Note that for some i and j, $w_G(i,j)$ is contributed exactly once, when i and j merge at $i \vee j$, and $n - |A| - |B|$ is the number of non-leaves at that step. Therefore: $\text{rev}_G(T) = \sum_{i,j \in V(G)} \text{mergerev}_G(i,j)$. In addition, note the contribution of each i,j pair, which is scaled by $w_G(i,j)$, is the number of leaves of $i \vee j$ for revenue, and the number of non-leaves of $i \vee j$ for cost. Therefore the contribution of each edge for revenue is n minus the contribution for cost, scaled by $w_G(i,j)$. In other words: $\text{rev}_G(T) = n \sum_{i,j \in V(G)} w_G(i,j) - \text{cost}_G(T)$.

While cost is a popular and well-founded metric, [Charikar and Chatziafratis, 2017] found that it is not constant factor approximable under the Small Set Expansion Hypothesis. On the other hand, [Moseley and Wang, 2017] proved that revenue is, and Average Linkage achieves a $1/3$-approximation. This makes it a more practical function to work with.

Our other function of interest is [Cohen-Addad et al., 2018]'s value function. This was introduced as a Dasgupta-inspired optimization function where edge weights represent distances.

It looks exactly like cost, except it is now a maximization function because it is in the distance context.

Definition 111 ([Cohen-Addad et al., 2018])**.** *The **value** of tree T on graph G with dissimilarity-based edge weights w_G is a maximization function.*

$$\text{val}_G(T) = \sum_{i,j \in V(G)} w_G(i,j) |\text{ leaves}(T[i \vee j])|.$$

Like revenue, value is constant factor-approximable. In fact, the best approximation (other than an SDP) for value is Average Linkage's $2/3$-approximation [Cohen-Addad et al., 2018]. To our knowledge, there are no distributable approximations for value.

11.2.3 Massively Parallel Communication (MPC)

Massively Parallel Communication (MPC) is a model of distributed computation used in programmatic frameworks like MapReduce [Dean and Ghemawat, 2008], Hadoop [White, 2009], and Spark [Zaharia et al., 2010]. MPC consists of "rounds" of computation, where parts of the input are distributed across machines with limited memory, computation is done locally for each machine, and then the machines send limited messages to each other. The primary complexities of interest are machine space, which should be $\widetilde{O}(n)$, and the number of rounds. Many MPC algorithms are extremely efficient. For instance, Affinity Clustering in some cases can have constantly many rounds, and otherwise may use up to $O(\log^2 n)$ rounds [Bateni et al., 2017].

11.3 Finding a k-sized maximum matching

The algorithm we introduce in Section 11.4 requires the use of a $(1-\epsilon)$-approximation for the maximum k-sized (or less) matching, where $k > n/2$. For this we will use [Ghaffari et al., 2018]'s $(1-\epsilon)$-approximation for maximum matching in MPC, which runs in $O(\log\log(n) \cdot (1/\epsilon)^{1/\epsilon})$ rounds with $O(n/polylog(n))$ space. Inspired by the results of [Hassin et al., 1997], we provide a distributed reduction between matching and k-matching. To do this, we add $n-2k$ vertices and edges of weight Q (which is found with a binary search) between the new and original vertices, and run the matching algorithm. Both the proof and algorithm (Algorithm 2) are found in the Appendix in the full paper.

Theorem 112. *There exists an MPC algorithm for k-sized maximum matching with nonnegative edge weights and max edge weight W for $k > n/2$ that achieves a $(1-\epsilon)$-approximation in $O(\log(nW)\log\log(n) \cdot (1/\epsilon)^{1/\epsilon})$ rounds and $O(n/polylog(n))$ machine space.*

11.4 Bounds on a matching-based hierarchical clustering algorithm

We now introduce our main algorithm, *Matching Affinity Clustering*. For revenue, we show it achieves a $\left(\frac{1}{3}-\epsilon\right)$-approximation for graphs with 2^N vertices, and a $\left(\frac{1}{9}-\epsilon\right)$-approximation in general. Similarly, for value, we show it achieves a $\frac{2}{3}\alpha$-approximation for graphs with 2^N vertices, and a $\frac{1}{3}\alpha$-approximation in general, given an α-approximation algorithm for minimum weighted k-sized matching in MPC.

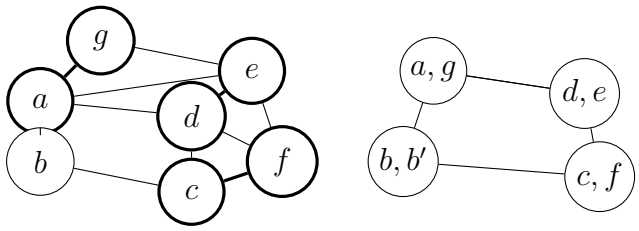


Figure 11.1: An example of the first iteration of Matching Affinity Clustering. We start by doing a 6-sized matching on the current graph on the right. We then duplicate unmatched vertices and merge to create the next cluster graph with 2^n vertices on the right. In subsequent iterations, matches are perfect. Edge weights are the Average Linkage between clusters (non-edges are zero).

11.4.1 Matching Affinity Clustering

Matching Affinity Clustering is defined in Algorithm 15. Its predecessor, Affinity Clustering, uses the MST to select edges to merge across, which sometimes causes imbalanced clusters. This is one reason why it cannot achieve a good approximation for revenue or value (Theorem 106). We fix this by, instead, using iterated maximum matchings (for similarity edge weights) and minimum perfect matchings (for dissimilarity edge weights). This ensures that on $n = 2^N$ vertices for some N, clusters will always be balanced.

The algorithm starts with one cluster per each of n vertices. Let 2^N be the smallest value such that $2^N \geq n$. It finds a maximum (resp. minimum) matching of size $k = 2n - 2^N$ (line 8, this means it matches $2n - 2^N$ vertices with $n - 2^{N-1}$ edges) and merges these vertices (line 12, Figure 11.1). Note that if $n = 2^N$, then $k = 0$, then the first step is a perfect matching. After this step, we have 2^{N-1} clusters. We then transform the graph into a graph of clusters with edge weights equal to the Average Linkage between clusters (lines 17-21). We find a maximum (resp. minimum) perfect matching of clusters in this new graph (line 10), then iterate.

Algorithm 15 Matching Affinity Clustering

Input A graph G with weight function $w : E(G) \to \mathbb{Z}^+$

$n \leftarrow |V|$
$N \leftarrow$ Such that $2^{N-1} < n \leq 2^N$
$\mathcal{C} \leftarrow G$ ▷ Current clustering graph, see Definition 114
while $n > 1$ **do**
    Yield $\mathcal{C}$ ▷ Output each level of the hierarchy
end while
if First iteration **then**
    $M \leftarrow \text{kMATCH}(\mathcal{C}, 2n - 2^N)$ ▷ Alg. 2 (Appendix)
else
    $M \leftarrow \text{MATCH}(\mathcal{C})$ ▷ [Ghaffari et al., 2018]
end if
$V \leftarrow \{v = (i, j) : (i, j) \in M\}$
$E \leftarrow V \times V$
$w \leftarrow \emptyset$
$n \leftarrow |V|$
Allocate each $C_j \in V$ to a machine
for Every machine m_j on C_j that merged $A_j, B_j \in V(\mathcal{C})$ **do**
    for Every other $C_k \in V$ that merged $A_k, B_k \in V(\mathcal{C})$ **do**
        $w(C_j, C_k) \leftarrow \frac{1}{4}(w_{\mathcal{C}}(A_j, A_k) + w_{\mathcal{C}}(A_j, B_k) + w_{\mathcal{C}}(B_j, A_k) + w_{\mathcal{C}}(B_j, B_k))$
    end for
end for
$\mathcal{C} \leftarrow (V, E, w)$

11.4.2 Revenue approximation

Now, we evaluate the efficiency and approximation factor of Matching Affinity Clustering with respect to revenue. In this section, edge weights represent the *similarity* between points. Proofs are in the Appendix in the full version of the paper. Ultimately, we will show the following.

Theorem 104. *In the revenue context (where edge weights are data similarity), with $\widetilde{O}(n)$ machine space, Matching Affinity Clustering achieves:*

- *a $(1/3 - \epsilon)$-approximation for revenue in $O(\log(n) \log\log(n) \cdot (1/\epsilon)^{O(1/\epsilon)})$ rounds when $n = 2^N$,*

- *and a* $(1/9 - \epsilon)$*-approximation for revenue in* $O(\log(nW)\log\log(n) \cdot (1/\epsilon)^{O(1/\epsilon)})$ *rounds in general.*

This will be a significant motivation for Matching Affinity Clustering's theoretical strength. As stated previously, one of the goals of Matching Affinity Clustering is to keep the cluster sizes balanced at each level. However, in the first step, note that Matching Affinity Clustering creates $n - 2^{N-1}$ clusters of size 2, and the rest of the vertices form singleton clusters. Therefore, to use this benefit of Matching Affinity Clustering, we need to ensure that cluster sizes will never deviate *too much*.

Lemma 113. *After the first round of merges, Matching Affinity Clustering maintains cluster balance (ie, the minimum ratio between cluster sizes) of* $1/2$.

After every matching, the algorithm creates a new graph with vertices representing clusters and edges representing the average linkage between clusters. We will call this a clustering graph.

Definition 114. *A* ***clustering graph*** $\mathcal{C}(G, C)$ *for graph* G *and clustering* $C = \{C_1, \ldots, C_k\}$ *of* G *is a complete graph with vertex set* $V = C$*. Its edge weights are the average linkage between clusters. Specifically, for vertices* v_{C_i} *and* v_{C_j} *in* $\mathcal{C}(G, C)$ *corresponding to clusters* C_i *and* C_j *where* $i \neq j$*, the weight of the edge between these vertices is:*

$$w_{\mathcal{C}(G,C)}(v_{C_i}, v_{C_j}) = \frac{1}{|C_i| \cdot |C_j|} \sum_{u \in C_i, w \in C_j} w_G(u, w).$$

The fact that the edge weights in the clustering graph are the average linkage between clusters denotes the similarities between Matching Affinity Clustering and Average Linkage. Essentially, we are trying to optimize for average linkage at each step, but instead of merging two

clusters, we merge many pairs of clusters at once with a maximum matching.

Since Matching Affinity Clustering computes this graph, we must show how to efficiently transform a clustering graph at the ith level, $\mathcal{C}(G, C^i)$ with clustering C^i, into a clustering at the $i+1$th level, $\mathcal{C}(G, C^{i+1})$ with clustering C^{i+1}.

Lemma 115. *Given $\mathcal{C}(G, C^i)$ and C^{i+1} where clusters are all composed of two subclusters in C^i, $\mathcal{C}(G, C^{i+1})$ can be computed in the MPC model with $\widetilde{O}(n)$ machine space and one round.*

This will eventually be used for our proof of efficiency of Theorem 104. For now, we return our attention to the approximation factor. Our approximation proof is going to observe the total merge cost and revenue across all merges on a single level of the hierarchy. For concision, we introduce the following notation to describe cost and revenue over a single clustering.

Definition 116. *The **clustering revenue** based off of some superclustering C' of C on graph G is the sum of the merge revenues of combining clusters in C to create clusters in C'. It is denoted by* $\text{clustering-rev}_G(C, C')$.

Definition 117. *The **clustering cost** based off of some superclustering C' of C on graph G is the sum of the merge costs of combining clusters in C to create clusters in C'. It is denoted by* $\text{clustering-cost}_G(C, C')$.

In order to prove an approximation for revenue, we want to compare each clustering revenue and cost. First, we must show that Matching Affinity Clustering has a large clustering revenue at any level.

Lemma 118. *Let clusters C^i and C^{i+1} be the ith and $i+1$th level clusterings found by Matching Affinity Clustering, where $C^0 = V$. Let p be the indicator that is 1 if n is not a power of 2. Then*

the clustering revenue of Matching Affinity Clustering at the ith level is at least:

$$\begin{aligned}&\text{clustering-rev}_G(C^i, C^{i+1})\\ &\geq 2^{3i-2p+1}\left(2^{n-i-1}-1\right)\sum_{(A,B)\in M_i} w_{\mathcal{C}(G,C^i)}(v_A, v_B).\end{aligned}$$

Now we address clustering cost. This time, we must show an upper bound for clustering cost at the ith level in terms of clustering revenue at the ith level. Let M_i be the matching Matching Affinity Clustering uses to merge C^i into C^{i+1}. Then M_i is a $(1-\epsilon)$-approximation of the optimum M_i^*.

Lemma 119. *Let C^i and C^{i+1} be the ith and $i+1$th level clusterings found by Matching Affinity Clustering, where the ith step uses matching $M_i \geq (1-\epsilon)M^*$ for maximum matching M^* and $C^0 = V$. Then the clustering cost of Matching Affinity Clustering at the ith level is at most:*

$$\begin{aligned}&\text{clustering-cost}_G(C^i, C^{i+1})\\ &\qquad\leq \frac{2^{2p+1}}{1-\epsilon}\,\text{clustering-rev}_G(C^i, C^{i+1}).\end{aligned}$$

Now we are ready to prove the approximation factor for Matching Affinity Clustering. We combine Lemma 119 with properties of revenue from Section 11.2.2 to obtain an expression for revenue in terms of $(n-2)$ times the sum of weights in the graph. We use this as a bound for the optimal revenue.

Lemma 120. *Matching Affinity Clustering obtains a $(1/3-\epsilon)$-approximation for revenue on graphs of size 2^N, and a $(1/9-\epsilon)$-approximation on general graphs.*

Finally, the round complexity is limited by the iterations and calls to the matching algo-

rithm. The space complexity is determined by the clustering graph construction.

Lemma 121. *Matching Affinity Clustering uses $\widetilde{O}(n)$ space per machine and runs in $O(\log(n) \log\log(n) \cdot (1/\epsilon)^{O(1/\epsilon)})$ rounds on graphs of size 2^N, and $O(\log(nW) \log\log(n) \cdot (1/\epsilon)^{O(1/\epsilon)})$ rounds in general.*

Lemmas 120 and 121 are sufficient to prove Theorem 104. Our algorithm achieves an approximation for revenue efficiently in the MPC model. In addition, the algorithm creates a desirably near-balanced hierarchical clustering tree.

We now prove the approximation bound tightness for Matching Affinity Clustering when $|V| = 2^N$. Recently, [Charikar et al., 2019a] proved by counterexample that Average Linkage achieves at best a $(1/3 + o(1))$-approximation on certain graphs. We find that Matching Affinity Clustering acts the same as Average Linkage on these graphs, and so has at best a $(1/3 + o(1))$-approximation.

Theorem 122. *There is a graph G on which Matching Affinity Clustering achieves no better than a $(1/3 + o(1))$-approximation of the optimal revenue.*

11.4.3 Value approximation

Now we consider Matching Affinity Clustering when edge weights represent *distances* instead of similarities. In this context, instead of running a k-sized maximum matching and then iterative general maximum matchings, we run a k-sized minimum matching and then iterative general minimum perfect matchings. Therefore, this algorithm is dependent on the existence of a k-sized minimum matching algorithm in MPC. Due to its similarity to other classical problems with $1 + \epsilon$ solutions in MPC [Ghaffari et al., 2018, Behnezhad et al., 2019e], we conjecture:

Conjecture 123. *There exists an MPC algorithm that achieves a $(1+\epsilon)$-approximation for minimum weight k-sized matching that uses $\widetilde{O}(n)$ machine space.*

Given such an algorithm, we can show that Matching Affinity Clustering approximates value.

Theorem 124. *Assume there exists an MPC algorithm that achieves an α-approximation for minimum weight k-sized matching in $O(f(n))$ rounds and $\widetilde{O}(n)$ machine space. In the value context (where edge weights are data distances) and in $O(f(n)\log(n))$ rounds with $\widetilde{O}(n)$ machine space, Matching Affinity Clustering achieves:*

- *a $\frac{2}{3}\alpha$-approximation for value when $n = 2^N$,*
- *and a $\frac{1}{3}\alpha$-approximation for value in general.*

The proof for this result is quite similar to the proof for the $2/3$-approximation of Average Linkage by [Cohen-Addad et al., 2018]. Instead of focusing on single merges, however, we observed the entire set of merges across a clustering layer in our hierarchy. Then we can make the same argument about the value across an entire level of the hierarchy, and use the cluster balance from Lemma 113 to achieve our result.

If Claim 123 hold, then the approximation factors become $2/3-\epsilon$ and $1/3-\epsilon$ respectively. We see a similar pattern as the revenue result, where the algorithm nears the state-of-the-art $2/3$-approximation achieved by Average Linkage [Cohen-Addad et al., 2018] on datasets of size $n = 2^N$, and still achieves a constant factor in general. Finally, we can additionally show the former approximation is tight. See the construction and proofs in the Appendix.

Theorem 125. *There is a graph G on which Matching Affinity Clustering achieves no better than a $(2/3+o(1))$-approximation of the optimal revenue.*

11.4.4 Round comparison to Affinity Clustering

In this section, we only consider Matching Affinity Clustering in the similarity edge weight context. The round complexities of Matching Affinity Clustering and regular Affinity Clustering depend on graph qualities, and in certain cases one outperforms the other. On dense graphs with n^{1+c} edges for constant c, [Bateni et al., 2017] showed that Affinity Clustering runs in $\lceil \log(c/\epsilon) \rceil + 1$ rounds. On sparse graphs, it runs in $O(\log^2 n)$ rounds, and it runs in $O(\log n)$ rounds when given access to a distributed hash table. We saw that Matching Affinity Clustering runs in $O\left(\log(n) \log\log(n) \cdot (1/\epsilon)^{O(1/\epsilon)}\right)$ rounds on graphs of size 2^N, and $O\left(\log(nW) \log\log(n) \cdot (1/\epsilon)^{O(1/\epsilon)}\right)$ in general for max edge weight W.

There are two situations where our algorithm outperforms Affinity Clustering. First, if the graph is sparse and the number of vertices is 2^N, then our algorithm runs in $O\big(\log(n) \log\log(n) \cdot (1/\epsilon)^{O(1/\epsilon)}\big)$ rounds, and Affinity runs in $O(\log^2(n))$ rounds. Otherwise, if the graph is sparse, Matching Affinity Clustering performs better as long as the largest edge weight is $W = o\left(\frac{\exp(\log^2(n)/\log\log(n))}{n}\right)$. This is strictly larger than constant. If W is large, Affinity Clustering is slightly more efficient. Finally, if the graph is dense, Affinity Clustering achieves an impressive constant round complexity, and is therefore more efficient. In any case, Matching Affinity Clustering is an efficient and highly scalable algorithm.

11.5 Experiments

We now empirically validate these results to further motivate Matching Affinity Clustering. The algorithm is implemented as a sequence of maximum or minimum perfect matchings, and the testing software is provided in supplementary material. The software as well as the five UCI

datasets [Dua and Graff, 2017] ranging between 150 and 5620 data points are exactly the same as those that were used for small-scale evaluation of Affinity Clustering [Bateni et al., 2017]. The data is represented by a vector of features. Similarity-based edge weights are the cosine similarity between vectors, and dissimilarity-based edge weights are the L2 norm. Most data and algorithms are deterministic and thus have consistent outcomes, but for any randomness, we run the experiment 50 times and take the average. Just like the evaluation of Affinity Clustering, our evaluation runs hierarchical clustering algorithms on k-clustering problems until we find a k-clustering within the hierarchy. This was compared to the ground truth clustering for the dataset.

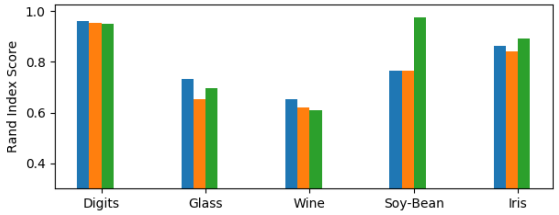
We evaluate performance using the Rand index, which was designed by [Rand, 1971] to be similar to accuracy in the unsupervised context. This is an established and commonly used metric for evaluating clustering algorithms and was used in the evaluation of Affinity Clustering.

Definition 126 ([Rand, 1971]). *Given a set $V = \{v_1, \ldots, v_n\}$ of n points and two clusterings $X = \{X_1, \ldots, X_r\}$ and $Y = \{Y_1, \ldots, Y_s\}$ of V, we define:*

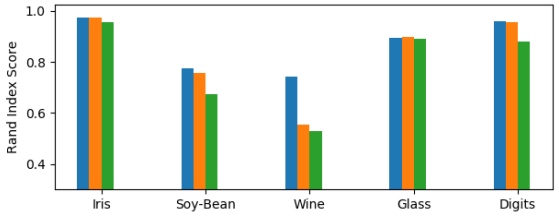
- *a: the number of pairs in V that are in the same cluster in X and in the same cluster in Y.*
- *b: the number of pairs in V that are in different clusters in X and in different clusters in Y.*

The Rand index $r(X, Y)$ is $(a + b)/\binom{n}{2}$. By having the ground truth clustering T of a dataset, we define the Rand index score of a clustering X to be $r(X, T)$.

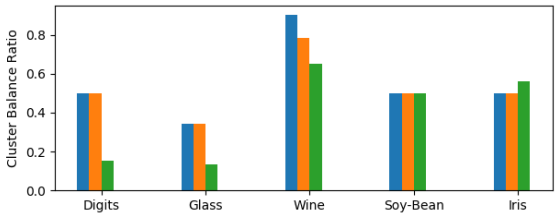
In addition, we are interested in evaluating the balance between cluster sizes in the clusterings, which indicates how good our algorithms are at evaluating balanced data. We use the *cluster size ratio* of a clustering, which was observed in [Bateni et al., 2017]. For a clustering $X = \{X_1, \ldots, X_r\}$, the size ratio is $\min_{i,j \in [r]} |X_i|/|X_j|$.



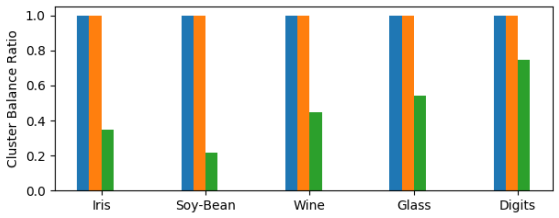
(a) Rand Index on Raw Data



(b) Rand Index on Filtered Data



(c) Cluster Balance on Raw Data



(d) Cluster Balance on Filtered Data

Figure 11.2: Rand Index and cluster balance on raw and filtered (randomly pruned for balance and $n = 2^N$) UCI datasets. Legend (bars, left to right): Max Matching Affinity Clustering is blue, Min Matching Affinity Clustering is orange, Affinity Clustering is green.

In Figure 11.2a, we see the Rand indices of Max Matching Affinity Clustering (ie, in the similarity context), Min Matching Affinity Clustering (ie, in the distance context), and Affinity Clustering. A Rand index is between 0 and 1, where higher scores indicate the clustering is more similar to the ground truth. Matching Affinity Clustering performs similarly to state of the art algorithms like Affinity Clustering on all data except the Soy-Bean dataset. A full evaluation on other algorithms (see the Appendix) illustrates that Matching Affinity Clustering outperforms other algorithms like Random Clustering and Average Linkage.

Figure 11.2b depicts the same information but on a slightly modified dataset. Here, we randomly remove data until (1) the dataset is of size 2^N, and (2) ground truth clusters are balanced. We did this 50 times and took the average results. This is motivated by Matching Affinity Clustering's stronger theoretical guarantees on datasets of size 2^N and ensured cluster balance. As expected, Matching Affinity Clustering performs consistently better than Affinity Clustering on filtered data, albeit by a a small margin in many cases. This shows that, experimentally, Matching Affinity Clustering performs better on balanced datasets of size 2^N.

Finally, Figures 11.2c and 11.2d depict the cluster size ratios on the raw and filtered data respectively. In theory, at every level in the hierarchy of Matching Affinity Clustering, no cluster can be less than half as small as another (Lemma 113). However, in our evaluation, we are comparing a single k-clustering, which may not precisely correspond to a level in the hierarchy. In this case, we take some clusters from the first level with fewer than k clusters and the last level with more than k clusters. Therefore, since cluster sizes double at each level, the lower bound for the cluster size ratio is now $1/4$. This is reflected Figure 11.2c, where Matching Affinity Clustering stays consistently above this minimum, and often exceeds it by quite a bit. On the filtered data (Figure 11.2d), Matching Affinity Clustering maintains perfect balance in every

instance, whereas Affinity Clustering performs much worse. Thus, Matching Affinity Clustering has proven empirically successful on small datasets.

11.6 Conclusion

Matching Affinity Clustering is the first hierarchical clustering algorithm to simultaneously achieve our four desirable traits. (1) Theoretically, it guarantees state-of-the-art approximations for revenue and value (given an approximation for MPC minimum perfect matching) when $n = 2^N$, and good approximations for revenue and value in general. Affinity Clustering cannot approximate either function. (2) Compared to Affinity Clustering, our algorithm achieves similar empirical success on general datasets and performs even better when datasets are balanced and of size 2^N. (3) Clusters are theoretically and empirically balanced. (4) it is scalable.

These attributes were proved through theoretical analysis and small-scale evaluation. While we were unable to perform the same large-scale tests as [Bateni et al., 2017], our methods still establish several advantages to the proposed approach. Matching Affinity Clustering simultaneously attains stronger broad theoretical guarantees, scalability through distribution, and small-scale empirical success. Therefore, we believe that Matching Affinity Clustering holds significant value over its predecessor as well as other state-of-the-art hierarchical clustering algorithms, particularly with its niche capability on balanced datasets.

11.7 Appendix

11.7.1 Affinity Clustering approximation bounds

In this section and all following sections, we provide the proofs for all theorems and lemmas introduced in this paper. It is broken down into sections based off of the sections corresponding to sections in the paper itself.

We start by proving Theorem 106. [Bateni et al., 2017] were in part motivated by the lack of theoretical guarantees for distributed hierarchical clustering algorithms. Thus, they introduced Affinity Clustering, based off of [Borůvka, 1926]'s algorithm for parallel MST. In every parallel round of Borůvka's algorithm, each connected component (starting with disconnected vertices) selects the lowest-weight outgoing edge and adds that to the solution, eventually creating an MST. Affinity Clustering creates clusters of each component. Note that Affinity Clustering was evaluated on a graph with weights representing dissimilarities between vertices, as opposed to our representation where weights are similarities. It is easy to verify that Affinity Clustering functions equivalently using max spanning tree in our representation. [Bateni et al., 2017] theoretically validate their algorithm by defining a cost function based off the cost of the minimum Steiner tree for each cluster in the hierarchy, however they do not motivate this metric. Therefore, it is more interesting to evaluate in terms of revenue and value. We ultimately show:

Theorem 106. *Affinity Clustering cannot achieve better than a $O(1/n)$-factor approximation for revenue or value.*

We will split this into two cases for each objective function. We start with revenue. First, note that when Affinity Clustering merges clusters in common connected components, it creates

one supercluster (ie, cluster of clusters) for all clusters in that component. Therefore, it may merge many clusters at once. A brief counterexample of why such a hierarchy does not work is when the max spanning tree is a star. Here, all vertices will be merged to a cluster in one round for a revenue of zero, which is not approximately optimal. To evaluate this algorithm, we must consider all possible ways Affinity Clustering might decide to resolve edges on the max spanning tree of the input graph. We propose a graph family that shows Affinity Clustering cannot achieve a good revenue approximation. The hierarchy we use for comparison is one that Matching Affinity Clustering would find, not including the k-matching step. We prove the following lemma.

Lemma 128. *There exists a family of graphs on which Affinity Clustering cannot achieve better than a $O(1/n)$-factor approximation for revenue.*

Proof. Consider a complete bipartite graph G with 2^n vertices such that each partition, L and R, has 2^{n-1} vertices, and all edges have weight 1. To make it a complete graph, we simply fill in the rest of the graph with 0 weight edges. We first consider how Affinity Clustering might act on G. To start, each vertex reaches across one if its highest weight adjacent edges, a weight one edge, and merges with that vertex. Therefore, a vertex in L will merge with a vertex in R, and vice versa. There are many ways this could occur. We consider one specific possibility.

Take vertices c_L and c_R from L and R respectively. It is possible that every vertex in $L \setminus c_L$ will merge with c_R and every vertex in $R \setminus c_R$ will merge with c_L. It doesn't matter which vertices c_R and c_L try to merge with. Then G is divided into two subgraphs, both of which are stars centered at c_L and c_R respectively. The spokes of the stars have unit edge weights, and all other edges have weight 0. In the next step, the two subgraphs will merge into one cluster. Since there are no non-leaves at that point, it contributes nothing to the total revenue. Therefore, all

revenues are encoded in the first step.

Since the subgraphs are identical, they will contribute the same amount to the hierarchy revenue. Recall that we are trying to prove a bound for whatever method Affinity Clustering might choose to break down a merging of a large subgraph into a series of independent clusters. Notice, however, due to the symmetries of the subgraphs, it does not matter in what order the independent merges occur. Therefore, we consider an arbitrary order. Let T be the hierarchy of Affinity Clustering with this arbitrary order. We must break it down into individual merges, and let T_0 be the portion of the hierarchy contributing to one of the stars. We only need to sum over the merges of nonzero weight edges.

$$\mathrm{rev}_G(T) = 2\,\mathrm{rev}_G(T_0).$$

At each step of merging the star subgraph, we merge a single vertex across a unit weight edge into the cluster containing the star center. Call this growing cluster C_i at the ith merge. Let v_i be the vertex that gets merged with C at the ith step. Then we can break this down into $2^{n-1} - 1$ total merges.

$$\begin{aligned}
\mathrm{rev}_G(T) =& 2 \sum_{i=1}^{2^{n-1}-1} merge\text{-}\mathrm{rev}_G(\{v_i\}, C_i), && (1)\\
=& 2 \sum_{i=0}^{2^{n-1}-2} 2^n - i - 1, && (2)\\
=& O(2^{2n}). && (3)
\end{aligned}$$

In step (1), we simply break a single star's merging into a series of individual merges in temporal order. Step (2) uses the fact that there are 2^n total vertices and i and 1 vertices in the groups being merged to apply the definition of merge revenue. Finally, in (3), we simply evaluate the summation.

Now we consider how Matching Affinity Clustering will act on this graph. It simply finds the maximum matching. In the first iteration, it must match across unit weight edges, and therefore is a perfect matching on the bipartite graph. After this, due to symmetry, it simply finds any perfect matching at each iteration until all clusters are merged. Since the number of vertices is 2^n, it can always find such a perfect matching. Let T' be Matching Affinity Clustering's hierarchy on G. We will break this down into clusterings at each level, as we did in Section 11.4. Note there are $\log_2(2^n) = n$ total clusterings required in the hierarchy. And at each clustering, we have a matching M_i that we merge across, so we can break it down into merges across matches. Let any C^i be the usual clustering at the ith level of this algorithm.

$$\begin{aligned}\text{rev}_G(T') &= \sum_{i=1}^{n} \text{clustering-rev}_G(C^i, C^{i+1}),\\ &= \sum_{i=1}^{n} \sum_{(A,B)\in M_i} \text{mergerev}_G(A, B).\end{aligned}$$

Note that by symmetry, each merge on a level contributes the same amount to the revenue. Therefore, we can simply count the number of merges and their contributions. At the ith level, there are 2^{n-i} total merges. The size of each cluster being merged is 2^{i-1}, so the number of non-leaves is $2^n - 2 \cdot 2^{i-1} = 2^n - 2^i$. Finally, the number of edges being merged across at each merge, since the graph is bipartite, is just 2^i.

$$\text{rev}_G(T') = \sum_{i=1}^{n} 2^{i-1}(2^n - 2^i) \cdot 2^i = O(2^{3n}).$$

Therefore, if we take the ratio of the revenues in the long run, we find the following.

$$\frac{\text{rev}_G(T')}{\text{rev}_G(T)} = O(2^n).$$

So by cleverly selecting n, we can make the clustering found by Affinity Clustering arbitrarily worse. If the size of the graph is $N = 2^n$, then Affinity Clustering can only achieve at best a $1/O(n)$ approximation for this graph.

□

We now move on to value.

Lemma 129. *There exists a family of graphs on which Affinity Clustering cannot achieve better than a $O(1/n)$-factor approximation for value.*

Proof. Consider simply a graph G that is a matching (ie, each vertex is connected to exactly one other vertex with edge weight 1) with $4n$ vertices. Again, recall that Affinity Clustering must match along the edges of a minimum spanning tree.

Partition the vertices into sets of four, which consist of two pairs. Consider one such set: v_1 is matched to v_2 and u_1 is matched to u_2. Most of the edges here are zero. Therefore, a potential component of the minimum spanning tree is the line v_1, u_1, u_2, v_2. If we do this for all sets, we can then simply pick an arbitrary root for each set (ie, v_1), make some arbitrary order of sets, and

connect the roots in a line. All of these added edges are weight 0, so this is clearly a valid MST.

However, note that each edge is contained within some set of four. So if Affinity Clustering merges across these edges first, then the largest cluster the 1-weight edges can be merged in has four vertices. Say the tree returned by Affinity Clustering is T. Note that we have $2n$ edges.

$$val(T) \leq 4 \sum_{i=1}^{2n} 1 = 8n.$$

We now observe the optimal solution. Since this is bipartite, we can simply merge each side of the partition first. Then we can merge the two partitions together at the top of the hierarchy. This means all edges are merged into the final, $4n$-sized cluster. Call this T^*.

$$val(T^*) = 4n \sum_{i=1}^{2n} 1 = 8n^2.$$

Thus $val(T)/val(T^*) = 1/O(n)$. Thus, Affinity Clustering cannot achieve better than a $1/O(n)$ approximation on this family of graphs. □

Finally, we simply combine the results of Lemma 128 and Lemma 129 to prove Theorem 106.

11.7.2 Distributed maximum $k$sized matching

In this section, we prove our results for distributed k-sized matching and additionally provide the pseudocode.

Theorem 112. *There exists an MPC algorithm for k-sized maximum matching with nonnegative edge weights and max edge weight W for $k > n/2$ that achieves a $(1-\epsilon)$-approximation in*

Algorithm 16 Approximate k-Sized Matching

1: Let U be set of dummy vertices for $|U| = n - 2k$
2: Let δ be the minimum of ϵ and the value satisfying $\epsilon = (c+1)\delta - \delta^2$ for $k \leq cn$
3: $V' \leftarrow V \cup U$ ▷ Constructing the transformed graph
4: $E' \leftarrow E \cup \{(u, v) : u \in U, v \in V\}$
5: **while** Binary search of $Q \in [nW]$ for the min Q that results in $|M| \leq k$ and $\frac{1}{k(1-\delta)} w(M) \leq Q$ **do**
6: $\quad w'(u, v) = Q$ for all $u \in U, v \in V$
7: $\quad M \leftarrow \text{MATCH}(V', E')$ ▷ [Ghaffari et al., 2018]'s matching algorithm
8: $\quad M \leftarrow M \setminus \{(u, v) : (u, v) \in M, u \in U, v \in V\}$ ▷ Remove edges not in G
9: **end while**

$O(\log(nW) \log\log(n) \cdot (1/\epsilon)^{1/\epsilon})$ *rounds and* $O(n/polylog(n))$ *machine space.*

Proof. Let us define Q by a binary search on values 1 through nW to find the minimum Q that satisfies a halting condition: that the resulting M the algorithm finds satisfies $|M| = k$ and $\frac{1}{k(1-\delta)} w(M) = Q$ (line 5) where W is the largest weight in G. First we transform the graph. Create a vertex set U of $n - 2k$ dummy vertices, add them to our vertex set, and connect them to all vertices in G with edge weights Q (lines 1 to 6). Then we run [Ghaffari et al., 2018]'s algorithm (line 7) on this new graph with error δ being the minimum of the value satisfying $\epsilon = (c+1)\delta - \delta^2$ and ϵ itself, and $k \leq cn$. Find the portion of this matching in G, and use this to check our halting condition.

We must start by showing that if $M_{G,k}$ is a $(1-\epsilon)$-approximate k-matching in G, then when $Q = \frac{1}{k(1-\epsilon)} w(M_{G,K})$, our algorithm finds a k-matching with a $(1-\epsilon)$-approximate weight. Assume there is such a matching, and consider when our algorithm selects Q for this value. Consider the matching the algorithm finds in the transformed graph. Assume for contradiction that some $u \in U$ is not matched to any $v \in V$, and every edge in the matching from G has weight over Q. Because u is not connected to any vertex in U, that means it isn't matched at all. And since u is connected to all $v \in V$ with a positive edge, for u to not have been matched, all $v \in V$

must have been matched. Since $k < n/2$, a perfect matching on G has at least k edges. Thus the weight of the algorithm's matching in G, $M_{\mathcal{A},G}$, is bounded below by k edges of weight greater than Q.

$$w(M_{\mathcal{A},G}) > kQ = \frac{1}{1-\epsilon} w(M_{G,k}).$$

But $\text{OPT}_{G,k}$ for k-sized matching in G must be at least as large as this, so then $(1-\epsilon)w(\text{OPT}_{G,k}) > w(M_{G,k})$, which is a contradiction on the assumption of $M_{G,k}$. Otherwise, if there is some $u \in U$ that is not matched to any $v \in V$ where one of the edges in the matching from G has weight Q or less, removing that match and pairing one of those vertices with u can only improve the matching. Thus we can add a processing step afterwards to ensure all $n - 2k$ new vertices are matched, while not decreasing the value of the matching.

Thus our algorithm matches all $n - 2k$ vertices in U to $n - 2k$ vertices in V, and the remaining $2k$ vertices in V create a matching for a total size of k or less. Thus the algorithm outputs an at most k-sized matching in G, so this selection of Q will make the halting condition true.

Let $M_{\mathcal{A}}$ be the matching our algorithm finds in the transformed graph. Then the total weight is,

$$w(M_{\mathcal{A}}) = (n-k)Q + w(M_{\mathcal{A},G}).$$

By the same argument as before, but without the $1 - \epsilon$ factor, there is an OPT in the transformed graph that matches all vertices in U to vertices in V. Thus the expression for OPT is similar, where $\text{OPT}_{G,k}$ is the optimum for a k-sized matching in G.

$$w(\text{OPT}) = (n-k)Q + w(\text{OPT}_{G,k}).$$

We know $M_{\mathcal{A}}$ is a $(1-\epsilon)$-approximation for OPT, so we can combine these two equations.

$$\begin{aligned}(n-k)Q + w(M_{\mathcal{A},G}) \geq &(1-\epsilon)(n-k)Q \\ &+ (1-\epsilon)w(\text{OPT}_{G,k}).\end{aligned}$$

We are interested in the portion of the solution in G, or $M_{\mathcal{A},G}$.

$$w(M_{\mathcal{A},G}) \geq -\epsilon(n-k)Q + (1-\epsilon)w(\text{OPT}_{G,k}).$$

Recall that $Q = \frac{1}{k(1-\epsilon)}w(M_{G,k})$, and $M_{G,k}$ is a k-sized matching in G. Therefore $w(M_{G,k}) \leq (\text{OPT}_{G,k})$. We can apply this to our inequality and simplify.

$$\begin{aligned}w(M_{\mathcal{A},G}) \geq &-\epsilon(n-k)\frac{1}{k(1-\epsilon)}w(\text{OPT}_{G,k}) \\ &+ (1-\epsilon)w(\text{OPT}_{G,k}), \\ &= \left(1-\epsilon(1+(n/k-1)(1-\epsilon))\right) w(\text{OPT}_{G,k}).\end{aligned}$$

Since $k = O(n)$, then n/k is bounded above by some constant. Then the approximation factor is $1-\epsilon(1+c(1-\epsilon)) = 1-(c+1)\epsilon+\epsilon^2$. So for any approximation factor ϵ, we can select some δ to run [Ghaffari et al., 2018]'s algorithm such that our algorithm gives a $(1-\delta)$-

approximation.

The algorithm searches for the minimum Q that satisfies this, so all that's left to prove is that a selection of $Q < \frac{1}{k(1-\delta)} w(M_{G,k})$ yields a k-sized matching $M_{\mathcal{A},G}$ where $Q = \frac{1}{k(1-\delta)} w(M_{\mathcal{A},G})$ still is a $(1-\epsilon)$-approximation. If this is true, it must have matched all $n-2k$ vertices in U with vertices in V and selected k edges from G. The value of this, where we sub in our value for Q, is:

$$\begin{aligned} w(M_{\mathcal{A}}) &= (n-2k)Q + w(M_{\mathcal{A},G}), \\ &= (n-2k)w(M_{\mathcal{A},G}) + w(M_{\mathcal{A},G}), \\ &= (n-2k+1)w(M_{\mathcal{A},G}). \end{aligned}$$

Therefore, the approximation factor on the transformed graph is equivalent to the approximation factor on G. Since we ran [Ghaffari et al., 2018]'s algorithm on the transformed graph with error δ where $\delta \leq \epsilon$, this yields a matching within error of $\text{OPT}_{G,k}$.

Therefore, our algorithm returns the desired approximation. This algorithm requires $O(\log(nW))$ iterations for the binary search on Q. In each iteration, the only significant computation in both round and space complexity is the use of the [Ghaffari et al., 2018] algorithm that uses $O(\log\log(n) \cdot (1/\epsilon)^{1/\epsilon})$ rounds and $O(n/polylog(n))$ machine space. Thus our algorithm runs in $O(\log(nW)\log\log(n) \cdot (1/\epsilon)^{1/\epsilon})$ rounds and $O(n/polylog(n))$.

□

11.7.3 Revenue approximation

Lemma 113. *After the first round of merges, Matching Affinity Clustering maintains cluster balance (ie, the minimum ratio between cluster sizes) of* $1/2$.

Proof. After the first round of merges, note that any duplicated vertex must be merged with its duplicate. This is because the edge weight between these vertices is essentially infinite (for the purposes of this paper, we will say it is arbitrarily large). Thus no duplicates will be matched with another duplicate, so each of the 2-sized clusters has at least one real vertex. In any subsequent merges, this property will hold. Thus it holds for all clusters beyond the initial singleton clusters.

□

Lemma 115. *Given* $\mathcal{C}(G, C^i)$ *and* C^{i+1} *where clusters are all composed of two subclusters in* C^i, $\mathcal{C}(G, C^{i+1})$ *can be computed in the MPC model with* $\widetilde{O}(n)$ *machine space and one round.*

Proof. We start by constructing the vertex set, V^{i+1}, which corresponds to the clusters at the new $i+1$th level. So for every $C_j^{i+1} \in C^{i+1}$, create a vertex $v_{C_j^{i+1}}$ and put it in vertex set V^{i+1}. It must be a complete graph, so we can add edges between all pairs of vertices.

Consider two vertices $v_{C_j^{i+1}}$ and $v_{C_k^{i+1}}$. Since we merge sets of two clusters at each round, these must have come from two clusters in C^i each. Say they merged clusters from vertices u_1, u_2 and v_1, v_2 respectively. Note that these vertices are from the previous clustering graph, $\mathcal{C}(G, C^i)$. Then the edges $(u_1, v_1), (u_1, v_2), (u_2, v_1), (u_2, v_2)$ have weights that are the average distances between corresponding i-level clusters (because they were from the previous clustering graph, $\mathcal{C}(G, C^i)$). Since the clusters in C^i all have the same size, we can calculate the weight as

follows.

$$w_{\mathcal{C}(G,C^{i+1})}(v_{C^{i+1}j}, v_{C_k^{i+1}}) = \frac{1}{4}(w_{\mathcal{C}(G,C^i)}(u_1, v_1) + w_{\mathcal{C}(G,C^i)}(u_1, v_2) + w_{\mathcal{C}(G,C^i)}(u_2, v_1) + w_{\mathcal{C}(G,C^i)}(u_2, v_2)).$$

This is true because the weights in $\mathcal{C}(G, C^i)$ are already the average weights in the ith level clusters, so they are normalized for the clusters size, which is $1/4$th of the cluster size at the next level. So when we sum together the four edge weights, we account for all the edges that contribute to the edge weight in the next level, then we only need to divide by 4 to find the average.

Matching Affinity Clustering can utilize one machine per $(i+1)$-level cluster . Each machine needs to keep track of the distance between its subclusters and all other subclusters at level i. It can then do this calculation to capture edge weights in one round with $O(n)$ space. □

Lemma 118. *Let clusters C^i and C^{i+1} be the ith and $i+1$th level clusterings found by Matching Affinity Clustering, where $C^0 = V$. Let p be the indicator that is 1 if n is not a power of 2. Then the clustering revenue of Matching Affinity Clustering at the ith level is at least:*

$$\text{clustering-rev}_G(C^i, C^{i+1}) \geq 2^{3i-2p+1}\left(2^{n-i-1}-1\right) \sum_{(A,B)\in M_i} w_{\mathcal{C}(G,C^i)}(v_A, v_B).$$

Proof. First, we want to break down the clustering revenue into the sum of its merge revenues.

Since each match in our matching M_i defines a cluster in the next level of the hierarchy, we can view each merge as a match in M_i. Then we apply the definition of merge revenue.

$$\begin{aligned}
&\text{clustering-rev}_G(C^i, C^{i+1}) \\
&\quad = \sum_{(A,B)\in M_i} \text{mergerev}_G(A, B), \\
&\quad = \sum_{(A,B)\in M_i} (2^n - |A| - |B|) \sum_{a\in A, b\in B} w_G(a, b).
\end{aligned}$$

Because we start with a power of two vertices after padding, each step can find a perfect matching, thus yielding a power of two many clusters of equal size at each level. Then since cluster size doubles each round, the cluster size at the ith iteration is 2^i. Even though some of the vertices may not contribute to the revenue, this is an upper bound on the size. So $n - |A| - |B|$ in this formula is at least $2^n - 2^{i+1}$. This is the work done in (1) below.

$$\begin{aligned}
&\text{clustering-rev}_G(C^i, C^{i+1}) \\
&\quad \geq (2^n - 2^{i+1}) \sum_{(A,B)\in M_i} \sum_{a\in A, b\in B} w_G(a, b), && (1) \\
&\quad = (2^n - 2^{i+1}) \sum_{(A,B)\in M_i} |A||B| w_{\mathcal{C}(G,C^i)}(v_A, v_B), && (2) \\
&\quad = 2^{2i-2p}(2^n - 2^{i+1}) \sum_{(A,B)\in M_i} w_{\mathcal{C}(G,C^i)}(v_A, v_B), && (3) \\
&\quad = 2^{3i-2p+1}(2^{n-i-1} - 1) \sum_{(A,B)\in M_i} w_{\mathcal{C}(G,C^i)}(v_A, v_B). && (4)
\end{aligned}$$

In (2), we simply substitute in place of the sum of the edge weights between A and B in G. By definition, the edge weight between v_A and v_B in the clustering graph is the average of the same

edge edge weights in G. Thus if we just scale that by $|A||B|$, we can substitute it in for the sum of those edge weights. In (3), we simply pull out $|A||B|$. These contain 2^i total vertices, and by Lemma 113, they contain at least 2^{i-1} that contribute to the revenue for a total factor of 2^{2i-2}. If there were 2^n vertices to start, then all clusters contain only real vertices, so the factor is 2^{2i}. With the indicator, this is 2^{2i-2p}. We then simplify in (4). □

Lemma 119. *Let C^i and C^{i+1} be the ith and $i+1$th level clusterings found by Matching Affinity Clustering, where the ith step uses matching $M_i \geq (1-\epsilon)M^*$ for maximum matching M^* and $C^0 = V$. Then the clustering cost of Matching Affinity Clustering at the ith level is at most:*

$$\begin{aligned}&\text{clustering-cost}_G(C^i, C^{i+1}) \\ &\qquad \leq \frac{2^{2p+1}}{1-\epsilon}\,\text{clustering-rev}_G(C^i, C^{i+1}).\end{aligned}$$

Proof of Lemma 119. As in Lemma 118, we want to break apart the clustering cost at the ith level into a series of merge costs. Again, we know the merge costs can be defined through the

matching M_i.

$$\begin{aligned}
&\text{clustering-cost}_G(C^i, C^{i+1}) \\
&\quad = \sum_{(A,B)\in M_i} \text{mergecost}_G(A,B), && (1)\\
&\quad = \sum_{(A,B)\in M_i} \Bigg(|A| \sum_{b\in B, c\notin A\cup B} w_G(b,c) \\
&\qquad\qquad + |B| \sum_{a\in A, c\notin A\cup B} w_G(a,c) \Bigg), && (2)\\
&\quad \le 2^i \sum_{(A,B)\in M_i} \Bigg(\sum_{b\in B, c\notin A\cup B} w_G(b,c) \\
&\qquad\qquad + \sum_{a\in A, c\notin A\cup B} w_G(a,c) \Bigg). && (3)
\end{aligned}$$

At this step, we broke the clustering cost into merge costs of the matching (1), applied the definition of merge cost (2), and pulled out $|A| = |B| \le 2^i$ (3). Consider the inner clusters. Instead of selecting $c \notin A \cup B$, we can consider c being in any other cluster from C^i. Let $C^i_1, \ldots, C^i_k \in C^i$ be all clusters other than A or B (i.e., $C^i_j \neq A, B$). Then we can define c as an element in any

cluster C_j^i for $j \in [k]$. After, we simply rearrange the indices of summation.

$$\begin{aligned}
&\text{clustering-cost}_G(C^i, C^{i+1}) \\
&\quad \leq 2^i \sum_{(A,B)\in M_i} \left(\sum_{b\in B, j\in[k]} \sum_{c\in C_j^i} w_G(b,c) \right. \\
&\qquad\qquad \left. + \sum_{a\in A, j\in[k]} \sum_{c\in C_j^i} w_G(a,c) \right), \\
&\quad = 2^i \sum_{(A,B)\in M_i} \left(\sum_{j\in[k]} \sum_{b\in B, c\in C_j^i} w_G(b,c) \right. \\
&\qquad\qquad \left. + \sum_{j\in[k]} \sum_{a\in A, c\in C_j^i} w_G(a,c) \right).
\end{aligned}$$

Recall that the edge weights in $\mathcal{C}(G, C^i)$ are the average edge weights between clusters in C^i on graph G. Again, to turn this into just the summation of the edge weights, we must scale by $|B||C_j^i|$ and $|A||C_j^i|$.

$$\begin{aligned}
&\text{clustering-cost}_G(C^i, C^{i+1}) \\
&\quad \leq 2^i \sum_{(A,B)\in M_i} \left(\sum_{j\in[k]} |B||C_j^i| w_{\mathcal{C}(G,C^i)}(v_B, v_{C_j}) \right. \\
&\qquad\qquad \left. + \sum_{j\in[k]} |A||C_j^i| w_{\mathcal{C}(G,C^i)}(v_A, v_{C_j^i}) \right), \\
&\quad = 2^{3i} \sum_{(A,B)\in M_i} \left(\sum_{j\in[k]} w_{\mathcal{C}(G,C^i)}(v_B, v_{C_j}) \right. \\
&\qquad\qquad \left. + \sum_{j\in[k]} w_{\mathcal{C}(G,C^i)}(v_A, v_{C_j}) \right).
\end{aligned}$$

For each iteration of the outer summation, we are taking all the edges with one endpoint as

A and all edges with one as B (besides the edge from A to B itself) and adding their weights. Since the only edge in M_i with an endpoint at A or B is the edge from A to B, the summation covers all edges with one endpoint as either A or B that are not in M_i. Consider an edge from some C to C' that isn't in M_i. In every iteration besides possibly the first, M_i matches everything, so we will consider C and C' in separate iterations of the sum. In both of these iterations, we add the weight $w_{\mathcal{C}(G,C^i)}(v_C, v_{C'})$. Thus, each edge in $\mathcal{C}(G, C^i)$ outside of M_i is accounted for twice, and no edge in M_i is accounted for.

Consider a multigraph H with vertex set $V(\mathcal{C}(G, C^i))$ and an edge set that contains all edges *except* those in M_i twice over. Note since $\mathcal{C}(G, C^i)$ was a complete graph, H must be a $2(|V(\mathcal{C}(G, C^i)| - 2)$-regular graph. Thus, we could find a perfect matching in H with a maximal matching algorithm, remove those edges to decrease all degrees by 1, and repeat on the new regular graph. Do this until all vertices have degree 0. Since each degree gets decremented by 1 each iteration, there must be a total of $2(|V(\mathcal{C}(G, C^i)| - 2)$ matchings $N_1, N_2, \ldots, N_{2(|V(\mathcal{C}(G,C^i)|-2)}$. Thus the clustering cost can be alternatively thought of as the sum of the weights of these alternate matchings in clustering graph $\mathcal{C}(G, C^i)$.

$$\begin{aligned}
&\text{clustering-cost}_G(C^i, C^{i+1}) \\
&\quad \leq 2^{3i} \sum_{j \in [2(|V(\mathcal{C}(G,C^i)|-2)]} w_{\mathcal{C}(G,C^i)}(N_j), && (4) \\
&\quad \leq 2^{3i} \sum_{j \in [2(|V(\mathcal{C}(G,C^i)|-2)]} w_{\mathcal{C}(G,C^i)}(M_i^*), && (5) \\
&\quad \leq 2^{3i+1}(|V(\mathcal{C}(G, C^i)| - 2) w_{\mathcal{C}(G,C^i)}(M_i^*). && (6)
\end{aligned}$$

In (4), we viewed the summations as the sum of weights of the alternative matchings described earlier. Step (5) utilizes the fact that M_i^* is a maximum matching, so the weight of any N_j is bounded above by the weight of M_i^*. Finally, in (6), we note that the summation does not depend on j, and so we remove the summation.

Since M_i is an approximation of the maximum matching on $\mathcal{C}(G, C^i)$, we know $w_{\mathcal{C}(G,C^i)}(M_i^*) \leq w_{\mathcal{C}(G,C^i)}(M_i)/(1-\epsilon)$. We can substitute this in and then rewrite it as the summation of edge weights in M_i.

$$\begin{aligned}
&\text{clustering-cost}_G(C^i, C^{i+1}) \\
&\qquad \leq 2^{3i+1}(|V(\mathcal{C}(G, C^i)| - 2)\frac{1}{1-\epsilon} w_{\mathcal{C}(G,C^i)}(M_i), \\
&\qquad \leq 2^{3i+1}(|V(\mathcal{C}(G, C^i)| - 2) \\
&\qquad\qquad \cdot \frac{1}{1-\epsilon} \sum_{(A,B)\in M_i} w_{\mathcal{C}(G,C^i)}(v_A, v_B).
\end{aligned}$$

The total number of vertices in $\mathcal{C}(G, C^i)$ (ie, the total number of clusters at the ith level) is just the total number of vertices over the cluster sizes: $2^n/2^i$. Plugging that in gives the desired

result.

$$
\begin{aligned}
&\text{clustering-cost}_G(C^i, C^{i+1}) \\
&\leq 2^{3i+1}\left(\frac{2^n}{2^i} - 2\right)\frac{1}{1-\epsilon}\sum_{(A,B)\in M_i} w_{\mathcal{C}(G,C^i)}(v_A, v_B), \qquad (7)\\
&\leq 2^{3i+2}\left(2^{n-i-1} - 1\right)\frac{1}{1-\epsilon}\sum_{(A,B)\in M_i} w_{\mathcal{C}(G,C^i)}(v_A, v_B), \\
&\leq \frac{2^{2p+1}}{1-\epsilon}\,\text{clustering-rev}_G(C^i, C^{i+1}). \qquad (8)
\end{aligned}
$$

The first steps consist of plugging in the cluster sizes and performing algebraic simplifications. Finally, Step (8) refers to Lemma 118 for the clustering revenue. Recall this is an upper bound for the clustering cost in G, and so the proof is complete.

So far, we have covered most of the lemma's claim. Now we just need to account for the first step, when we utilize the k-sized matching. Note the argument is dependent on M_i being a perfect matching, where M_0 may only be a maximum matching on $2N - 2^n$ vertices. The proof structure here will function similarly. In this case, we still construct a multigraph H as described on the subset of G containing vertices matching in M_0, then we add double copies of all the edges between vertices in the matching that aren't matched to each other for a max degree of $4N - 2^{n+1} - 4$ and thus create $4N - 2^{n+1} - 4$ matchings on the $2N - 2^n$ vertices to cover these edges. However, the cost also accounts for edges from the matched vertices to the unmatched vertices. We can construct a bipartite graph with all these edges once. Then all vertices on one side of the bipartition have degree $2^n - N$, and the vertices on the other side have degree $2N - 2^n$. So we can construct $2^n - N$ matchings with $2N - 2^n$ edges in this graph that cover all edges. Alternatively, this can be viewed as $2^{n+1} - 2N$ matchings on $2N - 2^n$ vertices. In this case,

we have a bunch of sized matchings on $2N - 2^n$ vertices, $N_1, N_2, \ldots, N_{2N-4}$. The rest of the arguments hold. Since $i = 0$, step (7) becomes the following.

$$\begin{aligned}
&\text{clustering-cost}_G(C^0, C^1) \\
&\leq 2\,(N-2) \cdot \frac{1}{1-\epsilon} \sum_{(A,B)\in M_i} w_{\mathcal{C}(G,C^i)}(v_A, v_B), && (9)\\
&\leq 2(2^n - 2)\frac{1}{1-\epsilon} \sum_{(A,B)\in M_i} w_{\mathcal{C}(G,C^i)}(v_A, v_B), && (10)\\
&\leq 4(2^{n-1} - 1)\frac{1}{1-\epsilon} \sum_{(A,B)\in M_i} w_{\mathcal{C}(G,C^i)}(v_A, v_B), \\
&\leq \frac{2^3}{1-\epsilon}\,\text{clustering-rev}_G(C^0, C^1). && (12)
\end{aligned}$$

This mirrors the computation in steps (7) through (8). In (9), we substitute i in for (7), and also replace the number of matchings with the new number of matchings (though recall at this point, we have already halved that value). Step (10) applies the fact that $N < 2^n$. Finally, in (11), we substitute in the clustering revenue. In the analysis for revenue, note that there do not exist unreal vertices yet, so we can consider $p = 0$ when we refer to the Lemma 118. However, for this Lemma proof, this is the case where we do eventually duplicate vertices, so we analyze it along with other clusterings where $p = 1$, so it only needs to meet the condition when $p = 1$.

$$\begin{aligned}
&\text{clustering-cost}_G(C^0, C^1) \\
&\qquad \leq \frac{2^{2p+1}}{1-\epsilon}\,\text{clustering-rev}_G(C^0, C^1). && (13)
\end{aligned}$$

Thus concludes our proof.

□

Lemma 120. *Matching Affinity Clustering obtains a* $(1/3 - \epsilon)$*-approximation for revenue on graphs of size* 2^N*, and a* $(1/9 - \epsilon)$*-approximation on general graphs.*

Proof. We prove this by constructing Matching Affinity Clustering. Our algorithm starts by allocating one machine to each cluster. Run Algorithm 16 for either the desired k- or $n/2$-matching, which finds our $1 - \epsilon$ approximate matching, to create clusters of two vertices each, then apply the algorithm from Lemma 115 to construct the next clustering graph based off this clustering. Repeat this process until we have a single cluster.

From Lemma 119, we see that at each round, the cumulative cost is bounded above by $\frac{2}{1-\epsilon}$ times the revenue. Then we utilize the definition of the cost of an entire hierarchy tree T to get

bounds.

$$\begin{aligned}
\text{cost}_G(T) &= 2 \sum_{u,v \in G, u \neq v} w_G(u,v) \\
&\quad + \sum_{\text{merges of } A,B} \text{mergecost}_G(A,B), && (1) \\
&= 2 \sum_{u,v \in G, u \neq v} w_G(u,v) \\
&\quad + \sum_{i \in [\log n]} \text{clustering-cost}_G(C^i, C^{i+1}), && (2) \\
&\leq 2 \sum_{u,v \in G, u \neq v} w_G(u,v) \\
&\quad + \frac{2^{3p+1}}{1-\epsilon} \sum_{i \in [\log n]} \text{clustering-rev}_G(C^i, C^{i+1}), && (3) \\
&\leq 2 \sum_{u,v \in G, u \neq v} w_G(u,v) + \frac{2^{2p+1}}{1-\epsilon} \text{rev}_G(T). && (4)
\end{aligned}$$

Step (1) simply break down the total cost into merge costs, and then step (2) consolidates merge costs in each level of the hierarchy into clustering costs. Note that every iteration halves the number of clusters, so there must be $\log n$ iterations. In (3), we apply the result from Lemma 119, and finally in (4), we add up all the clustering revenues into the total hierarchy revenue. We can then examine hierarchy revenue.

$$\begin{aligned}
\text{rev}_G(T) &= n \sum_{u,v \in G, u \neq v} w_G(u,v) - \text{cost}_G(T), \\
&\geq n \sum_{u,v \in G, u \neq v} w_G(u,v) - 2 \sum_{u,v \in G, u \neq v} w_G(u,v) \\
&\quad - \frac{2^{2p+1}}{1-\epsilon} \text{rev}_G(T).
\end{aligned}$$

The above simply utilizes the duality of revenue and cost, and then substitution from step (4). Next we only require algebraic manipulation to isolate $\text{rev}_G(T)$.

$$\frac{2^{2p+1}+1-\epsilon}{1-\epsilon}\,\text{rev}_G(T) \geq (n-2)\sum_{u,v\in G, u\neq v} w_G(u,v).$$

$$\text{rev}_G(T) \geq \frac{1-\epsilon}{2^{2p+1}+1-\epsilon}(n-2)\sum_{u,v\in G, u\neq v} w_G(u,v).$$

Then we know the optimal solution T^* can't have more than $n-2$ non-leaves, which means that each edge will only contribute at most $n-2$ times its weight to the revenue. Thus, $\text{rev}_G(T^*) \leq (n-2)\sum_{u,v\in G,u\neq v} w_G(u,v)$. In addition, since $\frac{1-\epsilon}{2^{2p+1}+1-\epsilon}$ can be arbitrarily close to $\frac{1}{2^{2p+1}+1}$, we rewrite it as $\frac{1}{2^{2p+1}+1}-\epsilon$. Apply all these to our most recent inequality to get the desired results.

$$\text{rev}_G(T) \geq \left(\frac{1}{2^{2p+1}+1}-\epsilon\right)\text{rev}_G(T^*).$$

For an input of size 2^n, we have $p=0$ and get a $\frac{1}{3}-\epsilon$ approximation for revenue. For all other inputs, $p=1$, and we get a $\frac{1}{9}-\epsilon$ approximation. We note that these applications of cost and revenue properties are heavily inspired by Moseley and Wang's proof for the approximation of Average Linkage [Moseley and Wang, 2017]. □

Lemma 121. *Matching Affinity Clustering uses $\widetilde{O}(n)$ space per machine and runs in $O(\log(n)\log\log(n)\cdot(1/\epsilon)^{O(1/\epsilon)})$ rounds on graphs of size 2^N, and $O(\log(nW)\log\log(n)\cdot(1/\epsilon)^{O(1/\epsilon)})$ rounds in general.*

Proof. First, we use Algorithm 16 to obtain a k-sized matching, which runs in $O(\log(nW)\log\log(n)\cdot(1/\epsilon)^{1/\epsilon})$ rounds and $O(n/polylog(n))$ machine space. After this, there are $\log n$ iterations, and at each iteration, we use [Ghaffari et al., 2018]'s matching algorithm Algorithm, which finds our

$(1-\epsilon)$-approximate matching in $O(\log\log n \cdot (1/\epsilon)^{O(1)})$ rounds and $O(n/polylog(n))$ machine space [Ghaffari et al., 2018]. We also transform the graph as in Lemma 115, which adds no round complexity, but requires $O(n)$ space. Thus in total, this requires $O(\log(nW)\log(n)\log\log(n) \cdot (1/\epsilon)^{O(1/\epsilon)})$ rounds and $O(n)$ space per machine. However, note that when $p = 0$, we can just run [Ghaffari et al., 2018]'s algorithm directly, and achieve a slightly better bound of $O(\log(n)\log\log(n) \cdot (1/\epsilon)^{O(1/\epsilon)})$ rounds and $O(n)$ space per machine. □

Theorem 122. *There is a graph G on which Matching Affinity Clustering achieves no better than a $(1/3 + o(1))$-approximation of the optimal revenue.*

Proof of Theorem 122. The graph G consists of N vertices. We divide the vertices into $N^{1/3}$ "columns" to make large cliques. In every column, make a clique with edge weights of 1. In addition, enumerate all vertices in each column. For some index i, we take all ith vertices in each column and make a "row" (so there are $N^{2/3}$ rows that are essentially orthogonal to the columns). Rows become cliques as well, with edge weights of $1 + \epsilon$. All non-edges are assumed to have weight zero.

This is the graph described by [Charikar et al., 2019a] to show Average Linkage only achieves a $1/3$-approximation, at best, for revenue. They are able to achieve this because Average Linkage will greedily select all the $1 + \epsilon$ edges to merge across first. We can then leverage these results by showing that Matching Affinity Clustering, too, merges across these edges first.

In our formulation, we assume $N = 2^{3n}$ for some n. Then there are 2^n columns and 2^{2n} rows with 2^{2n} and 2^n vertices respectively. Additionally, our algorithm skips the k-matching steps (ie, all vertices are real). In the first round, we can clearly find a maximum perfect matching by simply matching within the rows, and we can assure this for our approximate matching by

selecting a small enough error. In the next clustering graph, since edge weights are the average linkage between nodes, note that the highest edge weights are still going to be $1 + \epsilon$ within the rows. Therefore, this matching will continue occurring until it can no longer find perfect matchings within the rows. Since the rows are cliques of 2^n vertices, this will happen until all each row is merged into a single cluster. This is then sufficient to refer to the results of [Charikar et al., 2019a] to get a $1/3$ bound. □

11.7.4 Value approximation

Our goal in this section is to prove Theorem 124.

Theorem 124. *Assume there exists an MPC algorithm that achieves an α-approximation for minimum weight k-sized matching in $O(f(n))$ rounds and $\widetilde{O}(n)$ machine space. In the value context (where edge weights are data distances) and in $O(f(n)\log(n))$ rounds with $\widetilde{O}(n)$ machine space, Matching Affinity Clustering achieves:*

- *a $\frac{2}{3}\alpha$-approximation for value when $n = 2^N$,*
- *and a $\frac{1}{3}\alpha$-approximation for value in general.*

Since this is effectively the same algorithm as the revenue context, we can the analysis of Lemma 121 and simplify it to show the complexity of the algorithm. All that is left to do is prove the approximation. Our proof will be quite similar to that of [Cohen-Addad et al., 2018], however we will require some clever manipulation to handle many merges at a time. Fortunately, the fact that we merge based off a minimum matching with respect to Average Linkage makes the analysis still follow the [Cohen-Addad et al., 2018] proof quite well. We start with a lemma.

Lemma 139. *Let T be the tree returned by Matching Affinity Clustering in the distance context. Consider any clustering C at some iteration of Matching Affinity Clustering above the first level. Let C_i be the ith cluster which merged clusters A_i and B_i in the previous iteration. Say there are k clusters in C. Then given an α-approximation MPC algorithm for minimum weight k-sized matching:*

$$\frac{\sum_{i=1}^{k} w(A_i, B_i)}{\sum_{i=1}^{k} |A_i| \cdot |B_i|} \geq 2\alpha \frac{\sum_{i=1}^{k} (w(A_i) + w(B_i))}{\sum_{i=1}^{k} (|A_i|(|A_i| - 1) + |B_i|(|B_i| - 1))}$$

Proof. Let $a = \frac{1}{2}\sum_{i=1}^{k} |A_i|(|A_i| - 1)$ and $b = \frac{1}{2}\sum_{i=1}^{k} |B_i|(|B_i| - 1)$. Let $A = \cup_{i=1}^{k} A_i$ and $B = \cup_{i=1}^{k} B_i$. Using these, one can see that the average edge weight of all edges contained in any A_i or B_i cluster is:

$$\frac{\sum_{i=1}^{k} (w(A_i) + w(B_i))}{a + b}$$

These edges were all merged across at some point lower in the hierarchy. This means that the edge set between A_i's and B_i's are the union of all edges merged across in lower clusterings in the hierarchy. Therefore, by averaging, we can say there exists a clustering C' (with $|C'| = k'$) below C in the hierarchy, with clusters and splits similarly defined as C', A'_i and B'_i respectively, such that:

$$\frac{\sum_{i=1}^{k'} w(A'_i, B'_i)}{\sum_{i=1}^{k} |A'_i| \cdot |B'_i|} \geq \frac{\sum_{i=1}^{k} (w(A_i) + w(B_i))}{a + b}$$

Now we would like to build a similar expression for the edges between all A_i and B_i. The average

of these edge weights is the following expression:

$$\frac{\sum_{i=1}^{k} w(A_i, B_i)}{\sum_{i=1}^{k} |A_i| \cdot |B_i|}$$

Consider the iteration that formed C'. Notice because C is a higher level of the hierarchy, every cluster A'_j and B'_j must be a subset of some A_i or B_i. Fix some i, and consider all the edges that cross from some A'_j to some B'_k such that $A'_j, B'_k \subset A_i \cup B_i = C_i$. The union of all these edges precisely makes up the set of edges between A_i and B_i. Do this for every i, and we can see this makes up all the edges of interest. We can decompose this into a set of matchings across the entire dataset. By another averaging argument, we can say there exists another alternate clustering C'' (as opposed to C') which only matches clusters A'_j and B'_k that are descendants of A_i and B_i respectively such that:

$$\frac{\sum_{i=1}^{k} w(A''_i, B''_i)}{\sum_{i=1}^{k} |A''_i| \cdot |B''_i|} \leq \frac{\sum_{i=1}^{k} w(A_i, B_i)}{\sum_{i=1}^{k} |A_i| \cdot |B_i|}$$

Note that this was a valid matching, and therefore a valid clustering, at the same time that C' was selected. Also, note that either both C and C' were perfect matchings, or they were both restricted to the same k size in the first step of the algorithm. Thus, since C' was an α-approximate minimum (k-sized) matching in the graph where edges are the average edge weights between clusters, we know:

$$\frac{\sum_{i=1}^{k} w(A'_i, B'_i)}{\sum_{i=1}^{k} |A'_i| \cdot |B'_i|} \leq \alpha \frac{\sum_{i=1}^{k} w(A''_i, B''_i)}{\sum_{i=1}^{k} |A''_i| \cdot |B''_i|}$$

Putting this all together, we find the desired result:

$$\frac{\sum_{i=1}^{k} (w(A_i) + w(B_i))}{a + b} \leq \alpha \frac{\sum_{i=1}^{k} w(A_i, B_i)}{\sum_{i=1}^{k} |A_i| \cdot |B_i|}$$

□

And we can use this to prove our theorem, similar to the results of [Cohen-Addad et al., 2018].

Proof of Theorem 124. We prove this by induction on the level of the tree. At some level, with clustering C, consider truncating the entire tree T at that level, and thus only consider the subtrees *below* C, ie with roots in C. Call this tree T_C. We will consider the aggregate value accumulated by this level. Trivially, the base case holds. Then we can split the value of C into the value accumulated at the most recent clustering step and value one step below C. We use induction on the latter value. Since the approximation ratio is $\frac{1}{3}$ for $p = 1$ and $\frac{1}{2}$ for $p = 0$, we can write the ratio as $\left(\frac{1}{3}\right)^p \left(\frac{1}{2}\right)^{1-p}$.

$$\begin{aligned}
&\text{val}(T_C) \\
&\quad \geq \sum_{i=1}^{k} (|A_i| + |B_i|) w(A_i, B_i) \\
&\qquad + \left(\frac{1}{3}\right)^p \left(\frac{2}{3}\right)^{1-p} \sum_{i=1}^{k} (|A_i| w(A_i) + |B_i| w(B_i))
\end{aligned}$$

Now we would like to apply Lemma 1 to modify the first term in a similar manner to Cohen-Addad et al. Specifically, we want to extract terms of the form $|A_i|w(B_i)$ and $|B_i|w(A_i)$. We will find this is harder to do with our formulation of Lemma 1, and therefore we have to rely on Lemma x that says that the cluster balance is at least $\frac{1}{2}$. Let $m = \min\left(\{|A_i|\}_{i=1}^k \cup \{|B_i|\}_{i=1}^k\right)$ be the minimum cluster size. This and our cluster balance ratio implies that $m \leq |A_i|, |B_i| \leq 2m$ for all i. Note that when $N = 2^n$, we have perfect cluster balance, so $|A_i| = |B_i| = m$. Thus for our indicator p, $m \leq |A_i|, |B_i| \leq 2^p m$. Now we can manipulate our Lemma 1 result. Start by

isolating the numerator on the left.

$$\sum_{i=1}^{k} w(A_i, B_i) \geq 2\alpha \frac{\sum_{i=1}^{k} |A_i| \cdot |B_i| \sum_{i=1}^{k} (w(A_i) + w(B_i))}{\sum_{i=1}^{k} (|A_i|(|A_i| - 1) + |B_i|(|B_i| - 1))}$$

Note now that $\sum_{i=1}^{k} |A_i| \cdot |B_i| \geq m^2$ and $|A_i|(|A_i| - 1) \leq 2^{2p} m^2$ and similarly for B_i.

$$\begin{aligned}\sum_{i=1}^{k} w(A_i, B_i) \geq & 2\alpha \frac{km^2 \sum_{i=1}^{k} (w(A_i) + w(B_i))}{2^{2p+1} km^2} \\ = & \alpha \frac{\sum_{i=1}^{k} (w(A_i) + w(B_i))}{2^{2p}}\end{aligned}$$

To get the correct term on the left, we see that $\sum_{i=1}^{k} (|A_i| + |B_i|) w(A_i, B_i) \geq 2m \sum_{i=1}^{k} w(A_i, B_i)$. So we can multiply this result by $2m$, and then plug it into a portion of the first term. To preserve the ratio for both $p = 1$ and $p = 0$, we multiply it by $\left(\frac{2}{3}\right)^p \left(\frac{1}{2}\right)^{1-p}$.

$$\begin{aligned}\text{val}(T_C) \geq & \left(1 - \frac{2}{3}\right)^{1-p} \left(1 - \frac{1}{3}\right)^{p} \sum_{i=1}^{k} (|A_i| + |B_i|) w(A_i, B_i) \\ & + \frac{2m}{2^{2p}} \cdot \left(\frac{2}{3}\right)^{p} \left(\frac{1}{3}\right)^{1-p} \alpha \sum_{i=1}^{k} (w(A_i) + w(B_i)) \\ & + \left(\frac{1}{3}\right)^{p} \left(\frac{2}{3}\right)^{1-p} \alpha \sum_{i=1}^{k} (|A_i| w(A_i) + |B_i| w(B_i))\end{aligned}$$

Next, note $m \geq |A_i|, |B_i|$. This can be used on the second term.

$$\begin{aligned}
\text{val}(T_C) &\geq \left(\frac{1}{3}\right)^p \left(\frac{2}{3}\right)^{1-p} \sum_{i=1}^{k} (|A_i| + |B_i|) w(A_i, B_i) \\
&\quad + \cdot \left(\frac{1}{3}\right)^p \left(\frac{2}{3}\right)^{1-p} \alpha \sum_{i=1}^{k} (w(A_i) + w(B_i)) \\
&\quad + \left(\frac{1}{3}\right)^p \left(\frac{1}{2}\right)^{1-p} \alpha \sum_{i=1}^{k} (|A_i| w(A_i) + |B_i| w(B_i)) \\
&\geq \left(\frac{1}{3}\right)^p \left(\frac{2}{3}\right)^{1-p} \alpha \sum_{i=1}^{k} |C_i| w(C_i)
\end{aligned}$$

Therefore, this captures $\frac{2}{3}\alpha$ of the weight of each subtree at height C when $n = 2^N$, and $\frac{1}{3}\alpha$ more generally. □

Finally, we can show Theorem 125, which shows the tightness of the stronger approximation factor.

Theorem 125. *There is a graph G on which Matching Affinity Clustering achieves no better than a $(2/3 + o(1))$-approximation of the optimal revenue.*

Proof. Consider G which is almost a bipartite graph between partitions A and B (with $|A| = |B|$), except with a single perfect matching removed. For instance, if we enumerate $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_n\}$, we have $w(a_i, b_j) = 1$ for all $i \neq j$ and $w(a_i, b_i) = 0$. And since it's bipartite, $w(a_i, a_j) = w(b_i, b_j) = 0$ for all i, j.

Consider the removed perfect matching to get clusters $\{a_1, b_1\}, \ldots, \{a_n, b_n\}$. Matching Affinity Clustering could start by executing these merges, as this is a zero weight (and thus minimum) matching.

Now consider G', the remaining graph after these merges with a vertex for each cluster and edges representing the total edge weight between clusters. This is a complete graph of size n with 2-weight edges. By [Dasgupta, 2016]'s results, we know the value (with is calculated the same as cost) of any hierarchy on G' is $\frac{2}{3}(n^3 - n)$. However, this ignores the fact that clusters are size 2, so the contribution of this part to the hierarchy on G yields a revenue of $\frac{4}{3}(n^3 - n)$.

Now let's consider the obvious good hierarchy, where we simply merge all of A, then all of B, then merge A and B together. Thus all $n(n-2)$ edges will be merged into a cluster of size $2n$ for a total value of $2(n^3 - 2n^2)$.

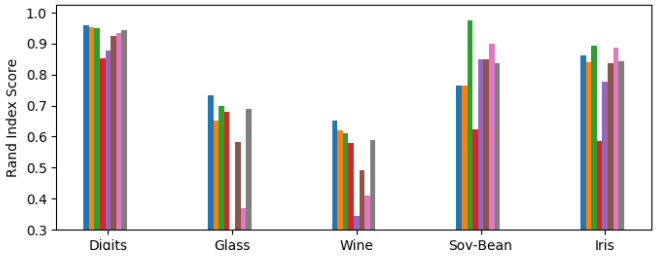
Asymptotically, then, Matching Affinity Clustering only achieves a ratio of $2/3$ on this graph. □

11.7.5 Experiments

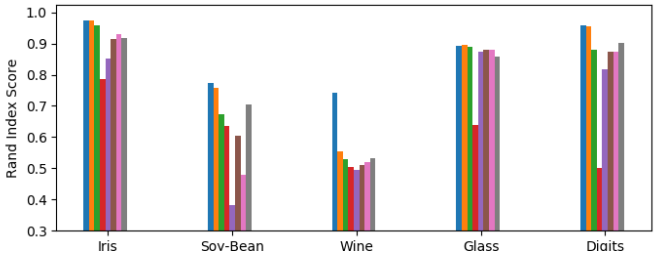
Here we include more complete visualizations of the performance of all tested algorithms. Like in the main body of text, we find the rand index and cluster size ratio on balanced and filtered data. These tests were run in the same way as the tests in the main body, we just present results on other common algorithms for completeness. The results are presented in Figures 11.3b through 11.3e.

Most of these results are as expected and simply reproduce the results from [Bateni et al., 2017]. However, we add one more algorithm: random clustering. Again, this is the clustering that randomly recursively partitions the data into a hierarchy. In our experiments, random clustering

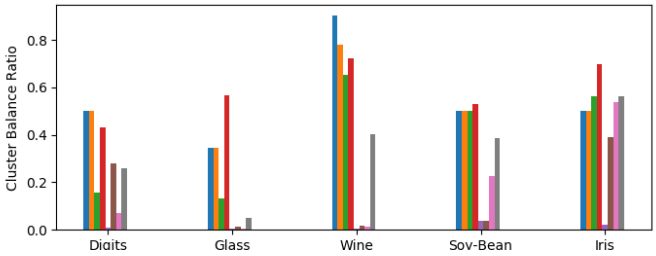
(a) Legend



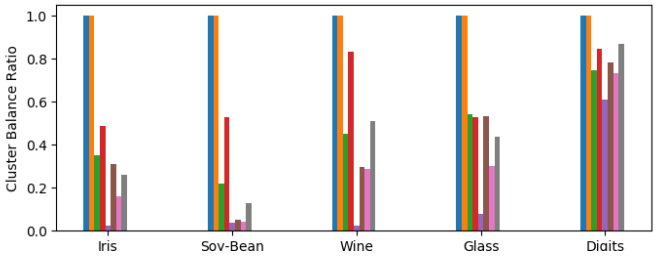
(b) Rand Index on Raw Data



(c) Rand Index on Filtered Data



(d) Cluster Balance on Raw Data



(e) Cluster Balance on Filtered Data

Figure 11.3: Rand Index scores and cluster balance on raw and filtered (randomly pruned so ground truth is balanced, $n = 2^N$) UCI datasets.

had surprisingly good cluster balance ratios (see Figure 11.3d). In fact, on raw data, it on average had more balanced clusters than Matching Affinity Clustering on three of the datasets.

There are three main observations about why Matching Affinity Clustering is still clearly more empirically successful than random clustering. First, notice that on filtered data in Figure 11.3e, Matching Affinity Clustering has more balanced clusters than Random clustering by a very wide margin. Second, it is clear in Figures 11.3b and 11.3c that Matching Affinity Clustering consistently and significantly outperforms random clustering. Third, random clustering is nondeterministic, whereas Matching Affinity Clustering is deterministic. Therefore, Matching Affinity Clustering's theoretical strengths and empirical performances are much stronger assurances than that of random clustering. Therefore, while an argument can be made that random clustering seems to empirically balance clusters well, Matching Affinity Clustering still does better in a number of respects, and thus is a more useful algorithm.

Chapter 12: Massively Parallel Tree Embeddings for High Dimensional Spaces

12.1 Introduction

Massive data-driven computation benefits greatly from embedding finite metric spaces into simpler spaces. Specifically, high-dimensional massive datasets, while often highly practical, are frequently too large to store on commodity hardware. Therefore, there is much interest in finding efficient methods for transforming this data into low-dimensional spaces. For instance, one of the most famous algorithms in high-dimensional geometry is the Johnson-Lindenstrauss transform [Johnson and Lindenstrauss, 1984], which embeds n points in the Euclidean space with any dimension into the $O(\log n)$-dimensional Euclidean space. Another branch of work solving this problem involves embedding metric spaces into tree metrics. A tree metric over n points is represented by an n-vertex tree, and therefore is also highly compact, requiring only $O(n)$ space. The main result of this paper is the first non-trivial massively parallel constant round extension of Bartal [Bartal, 1996]'s famous probabilistic tree metric embeddings of geometric datasets. We additionally provide a space-efficient massively parallel adaptation of the Johnson-Lindenstrauss transform.

Rabinovich and Raz [Rabinovich and Raz, 1998] showed that deterministically embedding a simple n-cycle into a tree metric requires $\Omega(n)$ distortion, or maximum proportional deviation between embedded and true distance. To circumvent this, Karp [Karp, 1989] leverages random-

ization to approximate a cycle by a path with low distortion. Alon, Karp, Peleg, and West [Alon et al., 1995] were the the first to probabilistically embed arbitrary metric spaces into trees, however they required up to $2^{O(\sqrt{\log n \log\log n})}$ distortion to do so. Bartal's work greatly surpassed this, achieving an $O(\log^2 n)$-approximation. A novel idea of Bartal's work in comparison with previous research is that it defines and utilizes *probabilistic partitions*, which ensures that two close points are more likely to be grouped together in the partition. By applying a hierarchy of probabilistic partitions, Bartal's algorithm embeds the input metric space into the so-called hierarchically well-separated tree (HST).

Tree embedding with HSTs has been improved a number of times since Bartal's inaugural work [Bartal, 1998, Charikar et al., 1998, Konjevod et al., 2001], culminating in the work of Fakcharoenphol, Rao, and Talwar [Fakcharoenphol et al., 2003], who improved the approximation factor to $O(\log n)$, notably yielding the first polylogarithmic approximation for the k-median problem. Since $\Omega(\log n)$ is also the lower bound [Bartal, 1996], this result sets a good foundation for expanding tree embeddings in other directions [Chan et al., 2005, Gupta et al., 2006, Gupta et al., 2003].

Metric tree embeddings have already been studied in PRAM [Blelloch et al., 2012,Friedrichs and Lenzen, 2018, Andoni et al., 2020], a classic model of parallel computing. Given a general metric space, Blelloch, Gupta, and Tangwongsan [Blelloch et al., 2012] designed a parallel $O(\log n)$-approximate metric tree embedding algorithm using $O(n^2 \log n)$ work (i.e., number of operations) and $O(\log^2 n)$ depth (i.e., parallel time). Friedrichs and Lenzen [Friedrichs and Lenzen, 2018] considered the shortest path metric given by a graph (i.e., graph metric) and gave a parallel $O(\mathrm{poly}(\log n))$-approximate metric tree embedding algorithm using $O(m+n^{1+\varepsilon})$ work and $O(\log n)$ depth where $\varepsilon > 0$ is an arbitrary constant and m is the number of edges of the in-

put graph. Andoni, Stein, and Zhong [Andoni et al., 2020] improved the work of [Friedrichs and Lenzen, 2018] to $(m+n)\cdot \text{poly}(\log n)$ though with a larger distortion, a high degree $\text{poly}(\log n)$.

Due to the success of many modern massively parallel systems such as MapReduce [Dean and Ghemawat, 2008], Hadoop [White, 2009], and Spark [Zaharia et al., 2016], a more refined model of parallel computing emerged — Massively Parallel Computation (MPC) [Karloff et al., 2010, Goodrich et al., 2011, Beame et al., 2017]— and has led to the development of new parallel algorithms in recent years. In this model, data is distributed to multiple machines where each machine has a sublinear amount of memory. We alternate between rounds of computation and rounds of communication where each machine can only send messages with size bounded by its local memory in a single round. Since communication is always the bottleneck of the model, the goal in MPC is to design an algorithm with few rounds (parallel time). We know that t-depth PRAM algorithms can be simulated in MPC in $O(t)$ rounds [Roughgarden et al., 2016]. Thus, in MPC, the simulation of any above mentioned PRAM algorithm would require $\Omega(\log n)$ parallel time. On the other hand, an $o(\log n)$-round MPC algorithm is always more desired in practice and faster MPC algorithms exist for many problems (see e.g., [Andoni et al., 2018, Andoni et al., 2019, Ghaffari and Uitto, 2019, Czumaj et al., 2018]).

Thus emerges the following natural question that we study in this paper:

Can we design an $o(\log n)$-round MPC algorithm for metric tree embedding?

The answer is no for general input metric spaces (e.g., the graph metric) with polylogarithmic distortion unless the 1-vs-2Cycle Conjecture [Yaroslavtsev and Vadapalli, 2018] is false. In geometric space, Arora [Arora, 1997]'s grid partitioning solves this in $O(1)$ MPC rounds with $O(\log^2 n)$ distortion.

We are the first to break this distortion barrier. On n points in $\mathbb{R}^d$ with aspect ratio $\text{poly}(n)$, there exists an $O(1)$-round MPC algorithm for $\widetilde{O}(\log^{1.5} n)$-approximate metric tree embedding.[1][2] This yields $O(1)$-round $\widetilde{O}(\log^{1.5} n)$-approximate MPC algorithms for Euclidean: minimum spanning tree, Earth-Mover distance, and densest ball.

We propose a new hierarchical probabilistic partitioning method to embed data in $\mathbb{R}^d$ into a tree with distortion $\widetilde{O}(\log^{1.5} n)$ using constant MPC rounds and low memory. Generally speaking, these methods iteratively partition the data and then recurse on each part in the partition until singletons or empty sets are reached. This yields a tree whose edges we weight and whose leaf set is the dataset. Its tree metric defines pairwise embedded distances. Our algorithm, *hybrid partitioning*, can be seen as a generalization of two existing partitioning methods: Arora [Arora, 1997]'s grid partitioning and Charikar et al. [Charikar et al., 1998]'s ball partitioning.

The main novelty of our methods is a hybridization of the two methods at each level of partitioning. Specifically, to partition the data, we group dimensions into r buckets, executing a ball partitioning on each bucket, and combining them with grid partitioning-like methods. If we set $r = 1$, all dimensions are in one bucket so the algorithm simply ball partitions the data. If $r = d$, the ball partitioning step simplifies greatly, and we end up effectively grid partitioning all points.

Our algorithm illustrates the trade-offs between the two methods in the parallel setting: grid partitioning methods reduce local memory and ball partitioning methods improve distortion. It turns out the key in our methods is to guarantee that an entire partition of the data can be stored in local memory. This becomes complicated using ball partitioning, since it requires a large number

[1] The aspect ratio of a point set is the ratio between the largest and the smallest interpoint distance.

[2] $\widetilde{O}(f(n))$ denotes $O(f(n) \cdot \text{polylog}(f(n)))$.

of attempts (and therefore, entire grids to store) to encode a partition. Our hybridization finds a nice way to reduce this space by only running ball partitions on subsets of dimensions.

Even with our space-reducing hybridization, a preprocessing application of the Johnson-Lindenstrauss transform to the input data is required to reduce dimensionality. Therefore, we include, as a result of independent and dependent interest, an efficient MPC implementation of the Fast Johnson-Lindenstrauss transform (Theorem 4). It achieves an $O(\log n)$-distortion embedding in $O(1)$ MPC rounds with low memory. The use of the fast transform over the original in particular allows for an important reduction in the total space for high-dimensional data.

12.1.1 Massively Parallel Computation

We work in the *Massively Parallel Computation* (MPC) model [Karloff et al., 2010, Beame et al., 2017]. MPC is an abstraction of MapReduce [Dean and Ghemawat, 2008] that models programming frameworks such as Hadoop [White, 2009], Spark [Zaharia et al., 2016], and Flume [Chambers et al., 2010]. MapReduce is used across industry, and is known for its fault tolerance and compatibility with commodity hardware. On graphs specifically, MPC has been used in many applications such as clustering [Bateni et al., 2017, Hajiaghayi and Knittel, 2020, Yaroslavtsev and Vadapalli, 2018] and Earth-Mover distance [Andoni et al., 2014], as well as theoretical problems like connectivity [Andoni et al., 2018, Andoni et al., 2019, Assadi et al., 2019c, Behnezhad et al., 2019d], matching and vertex cover [Ahn and Guha, 2015b, Assadi et al., 2019a, Behnezhad et al., 2019e, Ghaffari et al., 2018], minimum spanning tree [Andoni et al., 2014], and coloring [Assadi et al., 2019b, Behnezhad et al., 2019b]. Recent research has also explored adaptations of MPC [Behnezhad et al., 2019c, Behnezhad et al., 2020, Hajiaghayi et al.,

2022b, Hajiaghayi et al., 2022a, Roughgarden et al., 2016]. MPC is highly practical and for this reason, we study it in this work.

In MPC, the input is distributed across multiple machines. The computation proceeds in rounds, wherein each machine executes a local polynomial-time computation. At the end of the round, machines may send messages to and receive messages from any other machines. The total size of messages sent or received by a machine in a round is bounded by its local memory. MPC algorithm efficiency is measured by: the number of rounds (parallel time), the local memory, and the total space (number of machines times the local memory).

In this work, we consider MPC algorithms in the geometric context, where the input data contains n points in $\mathbb{R}^d$, represented by d-dimensional vectors. We use the most restrictive version of MPC where local space per machine is $O((nd)^\varepsilon)$ for any constant $\varepsilon \in (0, 1)$—termed the "fully scalable" regime [Andoni et al., 2018, Andoni et al., 2019]. All our algorithms are fully scalable, take $O(1)$ rounds, and use total space near linear in the input size $n \cdot d$.

12.1.2 Grid Partitioning Methods for Tree Metrics

We now describe two grid partitioning methods that we will extend in our work: random shifted grids and ball partitioning.

12.1.2.1 Random Shifted Grids

The first is the standard random shifted grid introduced by Arora [Arora, 1997]. Consider a geometric space in d dimensions. A random shifted grid is just a standard grid with cell width w whose origin is translated by some vector $(x_1, \ldots, x_d)$ where x_i is drawn uniformly at random

from $[0, w]$. Equivalently, each cell is translated by the vector $(x_1, \ldots, x_d)$. A visualization can be seen in Figure 12.1a.

Definition 141. *Given a cell width parameter w, consider a grid G of cell length w shifted randomly by a vector sampled uniformly at random from $[0, w]^d$. Place each point p into a partition representing the cell that contains it. This partitioning is a **random shifted grid partitioning** with scale w.*

We now discuss how to create a hierarchical partitioning from these random shifted grids. At this point, we will often refer to a *level*, which refers to a flat partitioning in a hierarchy, or alternatively, the recursive level in the hierarchical partitioning algorithm, starting with zero at the top. Let Δ be the aspect ratio (the maximum ratio between the maximum and minimum pairwise distances), $\mathcal{B}$ be a bounding box over our data (which we can say has width Δ), and ℓ be a parameter defining how many cells our grid should have, and how much it should increase at each level. We start by sampling a random shifted grid over $\mathcal{B}$ with cell width $w = \Delta/\ell$. Each point then falls into a cell in the grid. We create a partitioning of data where each partition corresponds to a non-empty cell such that it contains all points contained within the cell. We then recurse to make a hierarchical structure. The idea is to create a more refined grid (i.e., a grid with a smaller cell width) at each consecutive step. Generally, at the ith level in the hierarchy, partition each partition in the previous level using a randomly shifted grid of width Δ/ℓ^i. This then yields a new partitioning over our data, where partitions are more numerous and smaller, which we add to the hierarchy. For any partition, we stop partitioning as soon as it has one or no points.

The hierarchy defined by the random shifted grid partitioning procedure can be simply viewed as a tree. Let $\mathcal{B}$ be the root vertex. Then for each cell we create, add a vertex to the tree

with parent vertex corresponding to the cell's parent cell (i.e., the one which contains a superset of its points). Clearly this is a tree, and the leaves will either be empty (in that case, we can simply not create such a node) or they will represent a single datapoint. Therefore, we have created a tree structure to represent the data. Consider labeling each tree edge with weight $w\sqrt{d}$, where w was the cell width on that level. Then, the distance between two points is defined as the weight of the shortest path between the two points.

Grid partitioning is a nice, simple, classic technique that has inspired many results, including ball partitioning and our hybrid partitioning. It would be nice to simply use grid partitioning out-of-the-box, and it is not too hard to see that this can be implemented efficiently in MPC in $O(1)$ rounds with no significant local and total space issues. However, grid partitionings only achieve an $O(\log^2 n)$ distortion. We can do better.

12.1.2.2 Ball Partitioning

The ball parititioning method, depicted in Figure 12.1b, was introduced by [Charikar et al., 1998] for the purpose of derandomizing Bartal's algorithm. In spirit, it works quite similarly to random shifted grids, however we create partitions based off a grid of balls instead of the cells in a grid. In this method, we have two width parameters: the cell width and the ball radius. For simplicity of understanding, say that the ball radius is w and the cell width is $\ell = 4w$.

To create a single partitioning, first sample a random shifted grid of cell width $4w$ over the space. At each grid intersection point in our bounding box, create a ball of radius w. Note here that it is necessary that the cell width is more than twice as large as the ball radius, otherwise the balls will overlap and a point may fall into two balls. Even if the balls do not overlap, the resulting

partitioning will not necessarily partition the grid entirely, as some points may fall outside of all balls. To account for this, we simply continue to sample random shifted grids and create partitions for each grid, removing covered points as we go. We do this until all points are covered (or stop at some point and know that we succeed at covering all points with some probability).

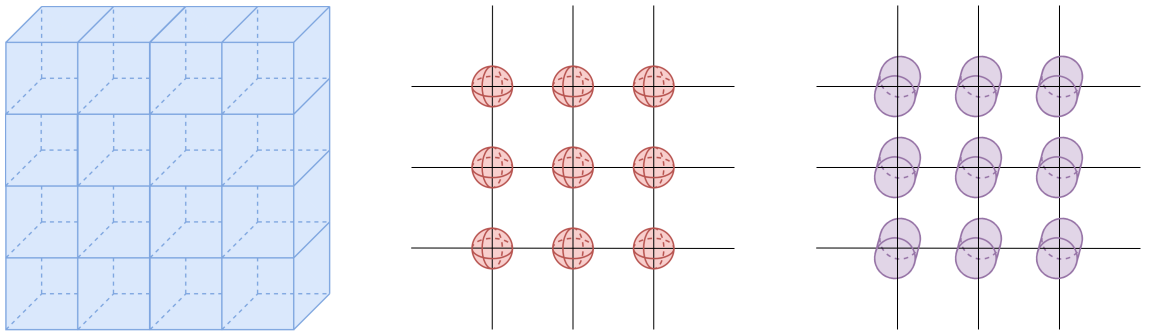
Definition 142. *Given a cell length parameter ℓ and radius w with $w = \frac{1}{4}\ell$, consider a sequence of grids $G_1, G_2, \ldots$ of cell length ℓ shifted randomly by vectors $s_1, s_2, \ldots$ sampled uniformly at random from $[0, \ell]^d$. Place a ball of radius w at each grid point for all $G_1, G_2, \ldots$. Place each point p into the first ball that contains it according to the grid ordering. This partitioning is a* ***ball partitioning*** *with scale w (or scale ℓ).*

The described method defines a partitioning at a single layer in the hierarchy. To create an entire hierarchy, we use the exact same strategy employed by the random shifted grids method, but instead creating our partitionings at each level using the ball partition.

Ball partitionings, while slightly more complicated, achieve a much nicer $O(\log^{1.5} n)$ distortion. The issue with this method is that it requires too much space to implement efficiently in MPC. Namely, we need to generate a large number of grids in order to cover the entire space, which will be exponential in d. Even though we reduce d to $O(\log n)$ using the Johnson-Lindenstrauss transform, this dependency is still too large. We later show in Lemma 153 how to reduce this dependency by adding more buckets of dimensions.

12.1.3 Our Contributions

We propose fully scalable, constant-round MPC algorithms for embeddings of geometric data in the MPC model. The set of points $P \subset \mathbb{R}^d$ is encoded as a set of d-dimensional vectors



(a) Grid partitioning, $w = 1$. (b) Ball partitioning, $w = 1/4$. (c) Hybrid partitioning, $w = 1/4$.

Figure 12.1: We depict one level (and one sample) of each discussed partitioning method. In grid partitioning (12.1a), we partition the grid into hypercubic cells of width 1 shifted by a random vector. In ball partitioning (12.1b), we place a ball of radius $1/4$ at each intersection of grid boundaries. Note that one instance of this placement is not sufficient to partition an entire space, as some parts are not covered by balls. Thus, we need to repeatedly draw randomly shifted grids and place balls at the intersections until every point in the space is covered by a ball. In hybrid partitioning (12.1c) with $r = 2$, we run a ball partitioning with ball radius $1/4$ on buckets of dimensions. If the z axis is sticking out towards the reader, then this involves two buckets: $\{x, y\}$ and $\{z\}$. We do a ball partitioning of the points projected onto the xy-plane and the z-axis independently, and then intersect them to get a partitioning. Since partitions in the xy plane are circles and partitions on the z axis are intervals, taking their intersection in 3-dimensional space results in cylindrical-shaped partitions.

(and therefore requires $O(nd)$ total space) and is assumed to have a bounded aspect ratio. Without loss of generality, we regard the coordinates of points as integers from $[\Delta] = \{1, 2, \cdots, \Delta\}$. For two points $x, y \in \mathbb{R}^d$, we use $\|x - y\|_2$ to denote their Euclidean distance. Our goal is to output a weighted tree containing all points in P such that $dist_T(p, q)$, the total length of the path from p to q on T (i.e., the tree metric on T), is close to $\|p - q\|_2$. Note that since the input size is $O(nd)$, $O((nd)^\epsilon)$ local space is considered fully scalable.

Our main result is the first fully scalable constant round MPC algorithm to break the $O(\log^2 n)$ expected distortion (i.e., the multiplicative deviation of $\mathbb{E}_T[dist_T(p, q)]$ from $||p-q||_2$) implied by Arora [Arora, 1997]'s grid partitioning. To our knowledge, other than the work of Arora, this is the only constant-round MPC algorithm for tree embeddings of high-dimensional data. Note that the success probability $1 - 1/\text{poly}(n)$ holds for any polynomial function in n.

Theorem 2. *Consider a set of n points $P \subseteq [\Delta]^d$ for $\Delta \in \mathbb{Z}_{\geq 1}$. There is an $O(1)$-round randomized MPC algorithm which computes a weighted spanning tree T over P when it succeeds, such that $\forall p, q \in P$,*

1. $dist_T(p, q) \geq \|p - q\|_2$,
2. $\mathrm{E}_T\left[dist_T(p, q)\right] \leq O(\sqrt{\log n} \cdot \log \Delta \cdot \sqrt{\log \log n}) \cdot \|p - q\|_2$.

The success probability is at least $1 - 1/\mathsf{poly}(n)$. The algorithm uses $O(n \cdot d + n \log n \cdot \left(\log \Delta \cdot \log \log n + \min(d, \log^2 n)\right))$ total space and each machine holds $O((nd)^{\varepsilon})$ local space for an arbitrary constant $\varepsilon \in (0, 1)$. If the algorithm fails, it reports failure.

The algorithm that achieves this result has two parts. The first is an efficient implementation of the *Fast Johnson Lindenstrauss transform*, a famous technique that reduces any high dimensional space into at most $O(\log n)$ dimensions. The second is a novel hybrid partitioning algorithm which combines Arora's random shifted grid partitioning [Arora, 1997] and Charikar et al.'s ball partitioning [Charikar et al., 1998] methods. On $O(\log n)$-dimensional data, this can be efficiently implemented in MPC. Together, these yield our main result.

This result stands in contrast to the results for general shortest-path metric of graphs. Conditioning on the 1-vs-2Cycle Conjecture [Yaroslavtsev and Vadapalli, 2018] (which postulates that distinguishing between a graph that is one n-cycle or two disjoint $n/2$-cycles requires $\Omega(\log n)$ MPC rounds), any fully scalable MPC algorithm for *general* graph connectivity needs $\Omega(\log n)$ rounds. If a graph is disconnected, then there are some $p, q \in P$ that are infinitely far apart. Any multiplicative approximation of the shortest path distance between p and q by a fully scalable MPC algorithm would approximate that distance as infinite, thus identifying the graph

as disconnected. It therefore requires $\Omega(\log n)$ rounds. This means that there is no multiplicative-approximate $o(\log n)$-round fully scalable MPC metric tree embedding algorithm for the graph metric under the 1-vs-2Cycle Conjecture [Yaroslavtsev and Vadapalli, 2018]. While we do not break this important barrier, our results show that the 1-vs-2Cycle Conjecture implies an infinite approximation gap for sublogarithmic MPC round shortest path distance in metric and geometric graphs.

12.1.3.1 Methods: Hybrid Partitioning

To achieve our result, we introduce the notion of *hybrid partitioning*, which combines two different geometric partitioning methods. Both partitioning methods are illustrated in Figure 12.1. The first is the standard random shifted grid introduced by Arora [Arora, 1997], where the data is partitioned by the cells of a grid, and the origin of the grid is offset by a random vector. The second is the randomized ball partitioning method, where the same random grid is used but instead of partitioning into the grid cells, balls of radius $1/4$ the width of the cells are placed at each line intersection [Charikar et al., 1998]. These define partitions. This is repeated until each point is covered.

The goal of a hybrid partitioning is to create an intermediate method which combines strategies from both partitioning algorithms. When parameterized to one extreme, hybrid partitioning is equivalent to grid partitioning. At the other extreme, it is equivalent to ball partitioning. We define hybrid partitioning with parameters $w, \ell \leq \Delta$ and $r \leq d$, where ℓ and w determine the scale of the partitions (similarly to ball partitioning) and r controls how to hybridize grid and ball partitioning. The following formal definition defines a flat partitioning of the space (and the

data). This can be made hierarchical by recursing on each partition. The resulting hierarchy is represented by a weighted tree: our embedding.

Definition 143. *In a d-dimensional space, consider bucketing all d dimensions into r* ***buckets*** *$\{\{1,\dots,d/r\},\{d/r+1,\dots,2d/r\},\dots\{d-d/r+1,\dots,d\}\}$ for parameter $r \le d$. Let ℓ be a scaling parameter and $w = \frac{1}{4}\ell$.[3] For an arbitrary point $p \in \mathbb{R}^d$, let $p^{(i)} \in \mathbb{R}^{d/r}$ be obtained from restricting (projecting) p on the dimensions in bucket i. For each bucket $1 \le i \le r$, run a ball partitioning on $P^{(i)} = \{p^{(i)} : p \in P\}$ with parameters w and ℓ. If a partitioning of $\mathbb{R}^d$ satisfies that p and q are in the same partition if and only if they are in the same partition for all buckets, we call it an r-**hybrid partitioning** with* ***scale*** *w (or scale ℓ).*

An illustration of hybrid partitioning on $\mathbb{R}^3$ can be seen in Figure 12.1c. Abstracting away the specific functionality of the algorithm, we can see the similarities between ball and grid partitioning, and how hybrid partitioning is an intermediate strategy. In this example, $r = 2$. If $r = 3$, hybrid partitioning must partition the space into cubes. If $r = 1$, it must partition the space into spheres.

We start with a sequential algorithm which is described in Section 12.3. Later, in Section 12.4, we will show how this algorithm can be implemented fully scalably in the MPC model, which results in our Theorem 2. We show that the sequential algorithm achieves the following guarantees:

Theorem 3. *Consider a set of n points $P \subseteq [\Delta]^d$ for $\Delta \in \mathbb{Z}_{\ge 1}$ and a parameter $r \in [d]$. Algorithm 17 computes a weighted spanning tree T over P such that $\forall p, q \in P$, $\|p-q\|_2 \le dist_T(p,q)$ and $\mathrm{E}_T[dist_T(p,q)] \le O(\sqrt{d \cdot r} \cdot \log \Delta) \cdot \|p-q\|_2$.*

[3] Without loss of generality, we assume d is divisible by r. Otherwise, we can concatenate 0s to each point to increase d to d' by a factor at most 2 and $d' \bmod r = 0$.

We now describe our sequential algorithm for hybrid partitioning. Without loss of generality, we suppose r divides d. We start by grouping the dimensions into r buckets each containing d/r dimensions. For each bucket, we project the data points into the space defined by these dimensions and then we run a ball partitioning (see Section 12.1.2.2 for a detailed description) with scale parameter $\Theta(w)$, meaning that the ball radius is w. Then each point is associated with one partition for each bucket. To join the buckets, we simply take the intersection of partitions over all buckets. For instance, two points p and q are in the same final partition under the hybrid partitioning if they are in the same partition obtained by the ball partitioning for every bucket.

To compute a tree embedding, we iteratively call the hybrid partitioning. At the beginning, $w = \Delta/2$. Once we obtain the partitions from the hybrid partitioning, we reduce the scale parameter w by a factor of 2 and recursively apply the hybrid partitioning on each partition. This yields a hierarchy whose root is the partition of all data points, and leaves represent a partitioning into singletons. As we move down the hierarchy, we connect each child node to its parent node with a weight proportional to $w \cdot \sqrt{r}$, where w is the scale parameter of the recursive *level*, and we show that $w \cdot \sqrt{r}$ is an upper bound of the diameter of a partition in the current level. Finally, we obtain a weighted tree. We refer readers to Section 12.3 for more details.

To see why this algorithm generalizes grid and ball partitioning, we consider extreme values of r. When $r = 1$, we have a single bucket. Clearly, then, our partitioning algorithm just executes the ball partitioning algorithm. When $r = d$, we partition each dimension separately. Therefore, the shapes of the intersections of all partitions end up being hypercubes. In our implementation, we let the ball radius be w and the cell length be $4w$, which means there is space between the hypercubes, unlike grid partitioning. However, if we instead let the ball radius be $w/2$, we eliminate all the uncovered space for $r = d$, and our algorithm becomes equivalent to

grid partitioning. Therefore, this is a hybridized version of the two partitioning methods. This algorithm also, notably, is implementable in the MPC model.

An interesting property of our hybrid partitioning is that we can find a bound on the cutting probability of two points (i.e., the probability that two points are assigned to different partitions) that is independent of r. This means that we can ensure some probability that two points are not separated at some level in the hierarchy regardless of the selection of r. Our bound for the diameter of partitions, however, is dependent on r. These two bounds are as follows. Note that the separation probability bound is only useful when $w > \sqrt{d}||p-q||_2$. Since w starts as Δ, this is true for at least one pair of points (though in most cases, many pairs) in the initial stages of the algorithm.

Lemma 144. *Consider a hybrid partitioning with parameters $w \in \mathbb{R}_{>0}$ and $r \leq d$ in the d-dimensional Euclidean space. For any two points $p, q \in \mathbb{R}^d$, the probability that p and q are assigned to different partitions is at most $O\left(\sqrt{d} \cdot \frac{\|p-q\|_2}{w}\right)$. If $p, q \in \mathbb{R}^d$ are assigned to the same partition, $\|p-q\|_2 \leq O(\sqrt{r} \cdot w)$.*

12.1.3.2 Methods: Johnson Lindenstrauss in MPC

In addition to our metric tree embedding result, we devise an efficient MPC implementation of the Fast Johnson-Lindenstrauss Transform (FJLT) [Ailon and Chazelle, 2006]. The FJLT utilizes sparse projections and the randomized Fourier transform to improve upon the standard Johnson-Lindenstrauss transform [Johnson and Lindenstrauss, 1984]. It reduces the dimension of data points to $O(\log n)$ with distortion at most $(1 \pm \xi)$ for any $\xi > 0$. The guarantee of our algorithm is the following, beating previous work in terms of total space by a factor of $\log n/\xi^2$:

Theorem 4. *Consider a set of points $P = \{p_1, p_2, \cdots, p_n\} \subset \mathbb{R}^d$. Let $\phi : \mathbb{R}^d \to \mathbb{R}^k$ be Fast Johnson Lindenstrauss Transform with $k = \Theta(\xi^{-2} \log n)$ for $\xi \in (0, 0.5)$. There is an MPC algorithm which outputs $\phi(p_1), \phi(p_2), \cdots, \phi(p_n)$ in $O(1)$ rounds. In addition, the total space of the algorithm is at most $O(nd + \xi^{-2} n \log^3 n)$ and each machine holds $O((nd)^\varepsilon)$ local space for an arbitrary constant $\varepsilon \in (0, 1)$.*

Details on the implementation and proofs can be found in Section 12.5. Along with being a separate interesting result in its own right, the ability to execute the transform in MPC is necessary for our main result. If we were to omit the transform as the first step before the hybrid partitioning, the hybrid partitioning would work on a potentially n-dimensional dataset. However, as we will see in the analysis for the MPC implementation of hybrid partitioning in Section 12.4, this would require an intractable computation. Specifically, the number of "balls" (i.e., partitions) at each partitioning is required to be exponential in d in order to cover the whole space with high probability. Since we must store the entire partitioning, potentially, this would make the total space exponential in n, which is quite excessive in MPC. However, we show that the dimensionality reduction of the Johnson Lindenstrauss transform is sufficient to yield an efficient MPC implementation.

We briefly note that the fast transform, as opposed to the original method, yields a more efficient MPC algorithm on high-dimensional data. Specifically, the standard transform would require $O(nd \log n)$ total space. However, if $d = \omega(\log^2 n)$, the rest of our algorithm (along with the fast transformation) achieves about $O(nd)$ total space. Therefore, we are able to achieve a total space reduction that is proportional to $\log n$ using the fast transform. In large-scale models such as MPC, even careful improvements like this can yield significant gains.

12.1.3.3 Applications

Both our metric tree embedding and fast Johnson Lindenstrauss results are useful for creating compact representations of high-dimensional geometric spaces. In fact, Theorem 2 is obtained by running the result from Theorem 4 to reduce the dimension of the data with constant distortion and then an MPC implementation of Algorithm 17 with $r = O(\log \log n)$ (Theorem 3).

Metric tree embeddings can be used to solve or approximate many problems in graph theory, and our results are no exception. This work can be extended to numerous applications, and we note three important ones below. Proofs can be found in the appendix. Note that an (α, β)-approximate densest ball is a bicriteria solution indicating that given a target diameter D, we approximate the problem of finding the ball that contains the most points within a factor of α with up to a β-multiplicative violation of the ball diameter. In other words, the ball may have diameter up to βD.

Corollary 145. *Consider a set of n points $P \subseteq \mathbb{R}^d$ with aspect ratio $\Delta \in \mathbb{Z}_{\geq 1}$. There is an $O(1)$-round randomized MPC algorithm which computes (on P with probability at least $1 - 1/\log \log n$) a:*

1. *$(1 - O(1/\log \log n), \widetilde{O}(\log^{1.5} n))$-approximate densest ball*
2. *$\widetilde{O}(\log^{1.5} n)$-approximate minimum spanning tree*
3. *$\widetilde{O}(\log^{1.5} n)$-approximate Earth-Mover distance*

It uses $O(n \cdot d + n \log n \cdot \left(\log \Delta \cdot \log \log n + \min(d, \log^2 n)\right))$ total space and each machine holds $O((nd)^{\varepsilon})$ local space for an arbitrary constant $\varepsilon \in (0, 1)$.

There are a few important things to note regarding these applications. First, to our knowledge, densest ball has not been studied in MPC and therefore our result is the first in this area. It is highly related to the problem of finding the densest subgraph of a given graph. In this problem the idea is to identify a subgraph H of an unweighted graph G that minimizes the density $d(H) = |E(H)|/|H|$, where $E(H)$ is the set of edges with both endpoints in H. While this has been studied in the sublinear regime [Bahmani et al., 2014, Ghaffari et al., 2019b], it does not imply any results for densest ball.

Additionally, a recent work [Chen et al., 2020a] proposed an efficient $\widetilde{\Theta}(\log n)$-approximate algorithm for EMD and MST in $\mathbb{R}^d$. However, their work is less broad since they directly compute EMD and MST, whereas we provide a general low-distortion embedding algorithm that can be used to solve a wide range of problems. In addition, there are applications where maintaining a space-efficient embedding of a dataset before computation may be highly practical. Therefore, our result is still of unique interest.

Finally, it is also notable that storing data on trees provides a unique structure for data computation. For instance, related works [Bateni et al., 2018a, Hajiaghayi et al., 2022b] introduced efficient low-memory MPC and AMPC algorithms for solving dynamic programs on trees.[4] Consider a problem that can be formulated on a tree embedding (i.e., where leaves correspond to the data set) with distortion α such that the problem can be approximated within a factor of $f(\alpha)$. Then we can apply these algorithms on top of our embedding to achieve an $f(\widetilde{O}(\log^{1.5}(n))$-approximation. Since the AMPC algorithm of [Hajiaghayi et al., 2022b] runs in constant rounds, then this process would require $O(1)$ rounds overall to embed and compute. Unfortunately, the

[4] *Adaptive* MPC, a related model where machines have adaptive in-round read-only access to a distributed hash table.

MPC algorithm of [Bateni et al., 2018a] requires $O(\log n)$ rounds, and therefore does not fully leverage our constant round complexity. Future work in this area may reveal more interesting results.

12.1.3.4 Related Lower Bounds

When the aspect ratio is $\text{poly}(n)$, the distortion of our current metric tree embedding is $\widetilde{O}(\log^{1.5} n)$. One of the future directions is to improve this approximation ratio in the MPC model. A natural goal would be to improve the distortion to $O(\log n)$ since any $o(\log n)$-distortion metric tree embedding would imply an embedding of the Earth-Mover distance into ℓ_1 with distortion better than the long-standing state-of-the-art embeddings [Naor and Schechtman, 2006] (even for planar Earth-Mover distance).

A number of related problems also exhibit similar apparent limitations. Embedding an n-point metric in ℓ_2 space into probabilistic trees needs at least $\Omega(\sqrt{\log n})$ distortion. This follows from a result of Rao [Rao, 1999] which states any finite planar metric of cardinality n, in particular a $\log n$-level diamond graph, can be embedded into ℓ_2 space with distortion $O(\sqrt{\log n})$ (which is tight according to Lee and Naor [Lee and Naor, 2004]) and another result of Gupta, Newman, Rabinovich, and Sinclair [Gupta et al., 2004] which states a $\log n$-level diamond graph needs distortion of at least $\Omega(\log n)$ to probabilistically embed into trees. Therefore, the distortion of embedding Euclidean points into trees must be at least $\Omega(\sqrt{\log n})$. While some of the above discussion does not enforce bounds on our problem, they are indicative of the difficulty of metric tree embedding in general. It is an open question to further explore this gap for high-dimensional Euclidean spaces.

12.2 Preliminaries

Here we introduce some preliminary definitions that will be useful in describing our algorithm and our results. This work provides tree embeddings on geometric data that exhibit two properties: they *dominate* the original geometric space and have small *distortion*. A metric space dominates another if the distance between any pairs of points is not smaller in the new metric space. This is a baseline assumption that many embeddings are shown to satisfy, as in Fakcharoenphol, Rao, and Talwar [Fakcharoenphol et al., 2003]. They define domination as follows.

Definition 146 ([Fakcharoenphol et al., 2003]). *A metric space $\mathcal{M}$ **dominates** another metric space $\mathcal{N}$ for all $q, p \in P$ for some set of points P if $d_{\mathcal{M}}(q,p) \geq d_{\mathcal{N}}(q,p)$, where $d_{\mathcal{M}}$ and $d_{\mathcal{N}}$ represent the distance function in each metric space respectively.*

Distortion (in our case, bi-Lipschitz distortion) measures the difference between the distance between two points in the original space and two points in a tree embedding. It is a measure of the goodness of the embedding in that low distortion implies we better approximate all pairwise distances, as measured by the largest proportional deviation.

Definition 147 ([Blelloch et al., 2012]). *A metric space $\mathcal{M}$ has α **distortion** over another metric space $\mathcal{N}$ if and only if $\mathcal{M}$ dominates $\mathcal{N}$ and for any points x and y, $d_{\mathcal{M}}(x,y) \leq \alpha d_{\mathcal{N}}(x,y)$, where $d_{\mathcal{M}}$ and $d_{\mathcal{N}}$ represent the distance function in each metric space respectively.*

If $\mathcal{M}$ is generated by a random process or represents a distribution over metric spaces, we instead view distortions in expectation, where $E[d_{\mathcal{M}}(x,y)] \leq \alpha d_{\mathcal{N}}(x,y)$. Specifically, our algorithm is randomized, so we will use this probabilistic interpretation of distortion. In this

paper, we develop algorithms that dominate the original geometric space with low distortion.

12.3 Hybrid Partitioning and its Distortion

Our sequential hybrid partitioning algorithm, Algorithm 17 (subroutines BuildGrids and BallPart can be found in the appendix, is a generalization of both Arora [Arora, 1997]'s random shifted grid and Charikar et al. [Charikar et al., 1998]'s random ball partition method. As we discussed in Section 12.1.2, the reason we cannot use either of these independently for a massively parallel algorithm is actually quite simple. For random shifted grids, the problem is that the resulting distortion is $O(\log^2 n)$. Our methods strive to achieve a better approximation factor than this. For ball partitions, the local space required is exponential in the dimension of the problem since we require many random shifted grids to cover all points in P. Even for d logarithmic in n, this method is unattainable in MPC.

Algorithm 17 Hybrid Partitioning: A Sequential Tree Embedding Algorithm

Input: Point set $P \subseteq [\Delta]^d$ and parameters $r \in [d]$, the number of buckets, and $U \in \mathbb{N}$, the number of grids

Output: T, a tree embedding of P

// Bucket the dimensions $[d]$ into r buckets
for $j \in [r]$ **do**
  $j_0 \leftarrow (d/r) \cdot (j-1)$
  $P^{(j)} \leftarrow \{p^{(j)} \mid p \in P : p^{(j)} = (p_{j_0+1}, p_{j_0+2}, \cdots, p_{j_0+d/r})\}$
end for

// Create a full ball partitioning and a corresponding hierarchy For each bucket
For all $j \in [r]$: $G^{(j)} = \mathsf{BuildGrids}(P^{(j)}, r, U)$
For all $j \in [r]$: $T_j \leftarrow \mathsf{BallPart}(P^{(j)}, G^{(r)})$,
If any ball partitionings failed, halt and report failure

// Join the partitionings to make a single, unified hierarchy
For all $j \in [r]$: $v_{0,j} \leftarrow$ the root of T_j
$v_0 = (v_{0,1}, v_{0,2}, \cdots, v_{0,r})$ // A vertex For the single cluster containing all of P
$T \leftarrow (\{v_0\}, \emptyset)$
Let $C : T \to 2^P$ where $C(v_0) = P$ // Identifying the cluster corresponding to a vertex
while $\exists v = (v_1, v_2, \ldots, v_r) \in \mathsf{leaves}(T)$ such that $|C(v)| > 1$ **do**
  $S = \mathsf{children}(v_1) \times \mathsf{children}(v_2) \times \cdots \times \mathsf{children}(v_r)$
  for $u = (u_1, u_2, \ldots u_r) \in S$ **do**
    $P_u \leftarrow \{p \in P : \forall j \in [r], p^{(j)}$ is in the cluster corresponding to $u_j\}$
    If $P_u \neq \emptyset$, add u to T as a child of v and set $C(u) = P_u$.
  end for
end while
Return T

The advantage of ball partitioning, however, is that it achieves a lower distortion. Therefore, the main idea of the hybrid partitioning method is to combine these two methods such that it is implementable in MPC, like random shifted grids, while still obtaining the improved distortion from ball partitioning. To do this, we introduce the notion of bucketing. To see how bucketing works, consider our bounding box $\mathcal{B}$ of our data, whose width is Δ, the aspect ratio of the data. To partition this bounding box, we start by bucketing the d dimensions into r buckets. That is, for a vector $\vec{x} = (x_1, \ldots, x_d)$, let it be bucketed as follows:

$$x^{(1)} = (x_1, \ldots, x_{d/r}), x^{(2)} = (x_{d/r+1}, \ldots, x_{2d/r}),$$

$$\ldots, x^{(r)} = (x_{d-d/r+1}, \ldots, x_d)$$

In a sense, the bucketing is taking a single point in space and projecting it into a number of different orthogonal subspaces. Since all dimensions are contained in exactly one subspace each, we do not lose any information in this process. This describes how one point is broken up into different projections. Over a set P of points, we create sets for each bucket of dimensions. The set contains the projection of each point in P into the respective bucket.

$$P^{(1)} = \{x^{(1)} | \forall x \in P\}, P^{(2)} = \{x^{(2)} | \forall x \in P\},$$

$$\ldots, P^{(r)} = \{x^{(r)} | \forall x \in P\}$$

At a high level, our algorithm, Algorithm 17, will create a ball partitioning on each bucket with cell width $4w$ and ball radius w for each level with different scale parameter w, and only group two points together if they are in the same partition for each bucket. Specifically, for each $i \in [r]$, let $\mathcal{C}_i$ be the partitioning created by the ball partitioning on only the dimensions in bucket i (i.e., the $(i-1)d/r+1$ through id/r dimensions of each data point). We create our $\mathcal{C}$ partitioning as follows: for each point p, to find the other points in its partition, let $\mathcal{C}^p$ be the set of partitions in $\mathcal{C}_1, \ldots, \mathcal{C}_r$ that contain p (note that some may be empty, as a ball partitioning may not cover p). Take the intersection of all partitions in $\mathcal{C}^p$ to form the partition C^p ($C^p = \bigcap_{C \in \mathcal{C}^p} C$). If C^p contains any other point, then it will be a partition in our new partitioning. Like in the ball partitioning method, we repeat this until all points are covered. If at any point in this method a

point does not have any other points within w of it, we simply partition it as its own partition.

Beyond this, the algorithm simply proceeds as the others to construct a hierarchy. Our ball radius starts as $w = \Delta/2$ for the top level and is scaled by $1/2$ at each recursive step. We recurse on each partition until all partitions are empty or singletons, and we create an edge of weight $\sqrt{r}w$ from each partition to its parent partition. With this hierarchy as our final tree embedding, it is not hard to see that the distance between any two points can be calculated by the number of levels of the hierarchy in which the two points are separated. If at any level they are separated, we add $\sqrt{r}w$ to their distance.

It is not too difficult to see how this method generalizes both the random shifted grid and ball partitions. Definitionally, if $r = 1$, then there is a single bucket containing all dimensions, and thus we are simply working in the original space. Then for any iteration of ball partitioning, two points are grouped together if they are captured by the same ball. Therefore, for $r = 1$, we are simply running the ball partitioning algorithm. If $r = d$ and if we let the ball radius be half the cell width, then each dimension is given its own bucket. When we run a ball partition on each bucket, or each 1-dimensional space, independently, we are just shifting that coordinate and partitioning that dimension into equally-sized intervals. Intersecting the partitions formed by partitioning dimensions like this is the same as defining d-dimensional cells in a grid based off a random shift vector composed of the random shifts of each individual dimension. Thus, we get precisely the random shifted grid.

Now that we have discussed the algorithm, we move on to the desired result. Note that all proofs can be found in the appendix. We ultimately show that, as long as we cover all points in each level (this is discussed probabilistically, by setting a high value of U, in Section 12.4):

Theorem 3. *Consider a set of n points $P \subseteq [\Delta]^d$ for $\Delta \in \mathbb{Z}_{\geq 1}$ and a parameter $r \in [d]$. Algorithm 17 computes a weighted spanning tree T over P such that $\forall p, q \in P$, $\|p - q\|_2 \leq dist_T(p, q)$ and $\mathrm{E}_T [dist_T(p, q)] \leq O(\sqrt{d \cdot r} \cdot \log \Delta) \cdot \|p - q\|_2$.*

The first property, the lower bound on the tree distance, is the notion of dominance. The second defines the amount of distortion resulting from the metric. Both are commonly required in popular probabilistic tree embeddings. It turns out that the first is much simpler to show for our algorithm:

Lemma 148. *For any two points $p, q \in P$, $\|p - q\|_2 \leq dist_T(p, q)$.*

The next goal is to prove the distortion. Crucially, to show this, we must find a relationship between the distance between two points and whether or not they are in the same partition at some level. This brings up the following lemma, which reveals an interesting property: the probability that two points are separated at a level can be bounded in a way that is independent of r, however the diameter of a partition relative to P is bounded by $\sqrt{r} \cdot w$.

Lemma 144. *Consider a hybrid partitioning with parameters $w \in \mathbb{R}_{>0}$ and $r \leq d$ in the d-dimensional Euclidean space. For any two points $p, q \in \mathbb{R}^d$, the probability that p and q are assigned to different partitions is at most $O\left(\sqrt{d} \cdot \frac{\|p-q\|_2}{w}\right)$. If $p, q \in \mathbb{R}^d$ are assigned to the same partition, $\|p - q\|_2 \leq O(\sqrt{r} \cdot w)$.*

Since this lemma is quite extensive, we break it down into two parts. The more interesting part is considering the probability that two points are separated in the partitioning at a certain level. We simply consider a ball partitioning:

Lemma 149. *Consider a ball partitioning over a set of points $P \subset \mathbb{R}^k$. For any two points $p, q \in P$, the probability that p and q are assigned to different partitions in a level with scale w is at most $O\left(\sqrt{k}\frac{\|p-q\|_2}{w}\right)$.*

In order to solve this, we consider a random variable $I_{q,p}$ that indicates if q and p exist in the same ball in a partitioning. This can be used to determine the probability of separation, but we can also relate it to the volume of the intersection of balls centered at these points with radius w. Eventually, we see that the probability of separation boils down to the probability that a random unit vector falls into the bound of a cap surface intersected with the surface of a sphere. In order to bound this, we need a slight detour into more pure geometric arguments, Lemma 150.

In particular, the following lemma shows that the probability that a random vector is near the equator of a unit sphere has a good upper bound.

Lemma 150. *Let $u \in \mathbb{R}^d$ be a random vector drawn uniformly from a unit sphere. Then for any $D, w \in \mathbb{R}_{\geq 0}$, we have*

$$\Pr[|u_1| \leq D/(2w)] = O\left(\sqrt{d} \cdot \frac{D}{w}\right).$$

While this lemma is close to the result we need, it unfortunately deals with vectors drawn from a voluminous shape, whereas we are interested in the probability that a random vector drawn from a surface falls within the bounds of intersected surfaces. A slight adaptation yields the following:

Lemma 151. *Let $v \in \mathbb{R}^d$ be a random vector drawn uniformly from a unit ball. Then for any*

$D, w \in \mathbb{R}_{\geq 0}$, we have

$$\Pr[|v_1| \leq D/(2w)] = O\left(\sqrt{d} \cdot \frac{D}{w}\right).$$

For two points to not be separated within some bucket in the partitioning, they must either be grouped together or left uncovered. This allows us to relate the probability of separation to the intersection of geometric surfaces. Ultimately, we can then derive a bound from Lemma 149 using Lemma 151 (see the proof of Lemma 149 in the appendix. Since two points separated on a level must be separated in some bucket, the separation probability is bounded by a union bound over the probability of separation in each bucket, yielding Lemma 144. As we have already showed the domination result, all we need is to show that the expected distortion is small, which is directly related to the probability that two points are separated on any level. This concludes Theorem 3.

12.4 Tree Embedding in MPC

In this section we show how to implement Algorithm 17 in $O(1)$ of rounds in the MPC model using the total space $O(n \cdot d + n \cdot \log \Delta \cdot \log n \cdot \log \log n)$. Specifically, we prove Theorem 2.

Theorem 2. *Consider a set of n points $P \subseteq [\Delta]^d$ for $\Delta \in \mathbb{Z}_{\geq 1}$. There is an $O(1)$-round randomized MPC algorithm which computes a weighted spanning tree T over P when it succeeds, such that $\forall p, q \in P$,*

1. $dist_T(p, q) \geq \|p - q\|_2$,
2. $\mathrm{E}_T\left[dist_T(p, q)\right] \leq O(\sqrt{\log n} \cdot \log \Delta \cdot \sqrt{\log \log n}) \cdot \|p - q\|_2$.

The success probability is at least $1 - 1/\mathsf{poly}(n)$. *The algorithm uses* $O(n \cdot d + n \log n \cdot \left(\log \Delta \cdot \log \log n + \min(d, \log^2 n)\right))$ *total space and each machine holds* $O((nd)^{\varepsilon})$ *local space for an arbitrary constant* $\varepsilon \in (0, 1)$. *If the algorithm fails, it reports failure.*

At a high level, our algorithm consists of four main steps:

1. Using our fast Johnson-Lindenstrauss (see Section 12.5), embed the data into $O(\log n)$ dimensions. This is the first $O(1)$ rounds of the algorithm.

2. We group dimensions into r buckets and generate grids for each bucket and distribute the grids and points among the machines. This requires 1 round on 1 machine.

3. We compute the hierarchical tree using ball partitioning. This requires 1 round of parallel computation.

4. For each node, we compute the path-to-root in the hybrid partitioning (to construct the final tree). This also requires 1 round of parallel computation.

In order to create the hierarchy in parallel, the grid of balls will be shared among all machines. Andoni [Andoni, 2009] lets us bounds the number of grids needed to cover the entire space:

Lemma 152 ([Andoni, 2009] Section 3.2.2)**.** *Consider a* d*-dimensional space* $\mathbb{R}^d$*, and fix some* $\delta > 0$*. Let* G *be a regular infinite grid of balls of radius* w *placed at coordinates* $4w \cdot \mathbb{Z}^d$*. Define* G_u*, for* $u \in \mathbb{N}$*, as* $G_u = G + s_u$*, where* $s_u \in [0, 4w]^d$ *is a uniformly selected random shift of the grid* G^d*. If* $U_d = 2^{O(d \log d)} \log 1/\delta$*, then the grids* $G_1, G_2, \ldots G_{U_d}$ *cover the entire space* $\mathbb{R}^d$*, with probability at least* $1 - \delta$*.*

Since we will be using grids to cover the whole space for each bucket and level we need the following lemma:

Lemma 153. *Consider n points over a d-dimensional space, and fix some parameter $\epsilon > 0, \delta > 0$. Define U as the number of grids of balls used in the hybrid partitioning. By setting $U = 2^{O((d/r)\log(d/r))} \cdot \log(\frac{r \log \Delta}{\delta})$, hybrid partitioning covers the whole space with probability at least $1 - \delta$.*

For standard ball partitioning the number of grids needed to cover the whole space would be too large. More specifically, for standard ball partitioning of a d-dimensional space we need $2^{O(d \log d)}$ grids. Even after reducing the data into $O(\log n)$ dimensions, this would result in $2^{O(\log n \log \log n)} = n^{O(\log \log n)}$ dimensions which would be too large considering we have $O(n^{\epsilon})$ space per machine. Hence the need to use *hybrid partitioning*.

To do so first we transform the data points from d-dimensional space to $O(\log n)$ dimensional space using our fast Johnson-Lindenstrauss transform. We fix $r = 2/\epsilon \cdot \log \log n$. Then we group the dimensions into r buckets and distribute them among the machines, each machine holding $\frac{n^{\epsilon}}{\log \Delta}$ points. This is because the ball partitioning needs $\log \Delta$ space per point to store the hierarchy, assuming we store the path from a vertex to the root.

For each bucket j we generate sets of grids $\{G_1^j, G_2^j, \cdots G_{log\Delta}^j\}$ and distribute them to the corresponding machines that hold points from this bucket. Where $G_i^j = \{B_1, B_2, \cdots B_U\}$ is the set of grids that cover the space in bucket j in level 2^i. Each B_k is a partitioning from a single randomly shifted instance of a ball partition.

Define U as the number of grids used to cover the space in our hybrid partitioning.

Lemma 154. *Consider n points over a d-dimensional space where $d = O(\log n)$, and a hybrid*

partitioning with $r = 2/\epsilon \cdot \log\log n$ buckets. The space used per machine to store the set of grids is $O(n^\epsilon)$ as long as $\delta = \Omega(1/\mathsf{poly}(n))$.

This proves all grids can be stored on each machine, and thus the local computation can proceed in constant rounds to complete Theorem 2.

Algorithm 18 MPC Hybrid Partitioning

Input: Point set $P \subseteq [\Delta]^d$ and a parameter $r \in [d]$ (after dimension reduction, e.g., see Section 12.5).
Output: T, a tree embedding of P

// Run on a single machine:
Split the dimensions $[d]$ into $r = O(\log\log n)$ buckets // As in the first step of Algorithm 17
Let $U = 2^{O((d/r)\log(d/r))} \cdot \log(r\log\Delta)$ // This comes from Lemma 153
For each $j \in [r]$: $G^{(r)} = \mathsf{BuildGrids}(P^{(j)}, r, u)$
SEnd G to all other machines
Partition P into parts P_i with $|P_i| = O(n^\epsilon/\log\Delta)$ and sEnd P_i to machine m_i

// In parallel:
for each machine m_i **do**
  For each $j \in [r]$: $T_j \leftarrow \mathsf{BallPart}(P_i^{(j)}, G^{(r)})$
  If any ball partitionings failed, halt and report failure
  for each $p \in P_i$ **do**
    for $j \in [r]$ **do**
      Find $path^j(p) \leftarrow (v_0^{(j)}, v_1^{(j)}, \ldots, v_{\log\Delta}^{(j)}, p^{(j)})$, the path from p to the root of T_j
    end for
    For $i \in [\log\Delta]$: $path_i(p) \leftarrow (v_i^{(1)}, v_i^{(2)}, \ldots, v_i^{(r)})$, the tuple For the ith element on p's path For all T_j
    $path(p) \leftarrow (path_1(p), path_2(p), \ldots, path_{\log\Delta}(p), p)$, the path from p to the root of T
  end for
  Let T_i be the union of $path(p)$ For all $p \in P_i$
  Return T_i as a part of the output tree T (implicitly, T is the union of all State Returned T_is)
end for

12.5 MPC Fast Johnson-Lindenstrauss

The Johnson Lindenstrauss transform, which maps any high-dimensional data into logarithmic dimensions with arbitrarily small distortion constant, is a foundational method for (potentially) significant reductions in data dimension in the sequential setting. Unfortunately, in

the massively parallel setting, this effectively becomes a general matrix multiplication problem, which does not meet the round and space complexity goals of many MPC algorithms. Particularly, we can achieve this either using $O(ndk) = O(nd \log n)$ total space in a constant number of rounds [Epasto et al., 2021] or using linear space $O(nd)$ in $O(\log n)$ rounds.

To prove our main result, Theorem 2, we require first applying the Johnson Lindenstrauss transform to reduce the dimensionality of our data, then we apply our methods from Section 12.4. The transform must be applied first because otherwise the number of balls required to partition the entire space with sufficiently high probability is too large, and they cannot be fit within our total space requirements.

Recall that our final results manage to shave off the additional logarithmic factor in the total space. Namely, running an $O(nd \log n)$ total space implementation of the Johnson Lindenstrauss transform would increase our total space. Therefore, we propose a solution to the dimension-reducing problem in the MPC model based on a more recent iteration of the Johnson Lindenstrauss transform by Ailon and Chazelle [Ailon and Chazelle, 2006] that uses linear total space.

Consider a dataset of n points in a d-dimensional Euclidean space. The Fast Johnson Lindenstrauss Transform (FJLT), developed by Ailon and Chazelle [Ailon and Chazelle, 2006], is a quickly computed transformation that maps the input points into a k-dimensional space while preserving distances within a $(1 \pm \xi)$ factor. Here, $k = c\xi^{-2} \log n$ for some constant c.

They show that the FJLT, $\phi(x) = k^{-1}PHDx$, for some input x can be computed relatively quickly. Matrices P, H, and D are designed as follows:

1. P is a $k \times d$ matrix with $P_{ij} = 0$ with probability $1 - q$, otherwise it is sampled from the

Gaussian distribution $N(0, q^{-1})$. We let q take value:

$$q = \Theta\left(\min\left(\frac{\log^2 n}{d}, 1\right)\right)$$

2. H is a normalized $d \times d$ Walsh-Hadamard matrix. In other words, it is the d-dimensional discrete Fourier transform. In particular, $H_{i,j} = d^{-1/2} \cdot (-1)^{\langle i-1, j-1\rangle}$ where $\langle i-1, j-1\rangle$ is the bitwise inner product of the binary representations of $i-1$ and $j-1$.

3. D is a random diagonal $d \times d$ matrix where $D_{i,i} = -1$ with probability 0.5 and $D_{i,i} = 1$ otherwise.

We now propose a method to parallelize this algorithm in the MPC model as shown in Algorithm 19. We consider the case where each machine has local space $O((nd)^{\varepsilon})$ for some arbitrary constant $\varepsilon \in (0, 1)$. The algorithm takes an input A which is a $d \times n$ matrix. To compute $PHD(A)$, we simply apply one matrix at a time. To compute $D(A)$, since D is diagonal, we can use $O(1)$ rounds and $O(d)$ total space to generate all $D_{i,i}$. To multiply D with A, since D is diagonal, each of the resulting nd entries in DA form a simple binary multiplication problem. We simply allocate $(nd)^{1-\epsilon}$ machines to do $O((nd)^{\epsilon})$ computations each. To apply H, we can simply apply the fast Fourier transform in the MPC model introduced by Hajiaghayi, Saleh, Seddighin, and Sun [Hajiaghayi et al., 2021]. This step takes $O(1/\varepsilon)$ rounds. Finally, we generate P in a similar way to D, by allocating $O((nd)^{\epsilon})$ entries per machine and randomly generating numbers. To apply P, we can bound the total number of multiplications by the results of Ailon and Chazelle, and then distribute additions across machines iteratively to compute the matrix in $O(1/\varepsilon)$ rounds.

Theorem 4. *Consider a set of points $P = \{p_1, p_2, \cdots, p_n\} \subset \mathbb{R}^d$. Let $\phi : \mathbb{R}^d \to \mathbb{R}^k$ be Fast Johnson Lindenstrauss Transform with $k = \Theta(\xi^{-2} \log n)$ for $\xi \in (0, 0.5)$. There is an MPC algorithm which outputs $\phi(p_1), \phi(p_2), \cdots, \phi(p_n)$ in $O(1)$ rounds. In addition, the total space of the algorithm is at most $O(nd + \xi^{-2} n \log^3 n)$ and each machine holds $O((nd)^\varepsilon)$ local space for an arbitrary constant $\varepsilon \in (0, 1)$.*

Proof. Let $A \in \mathbb{R}^{d \times n}$ be the concatenation of points $p_1, p_2, \cdots, p_n$. Then, the goal is to compute $P \cdot H \cdot D \cdot A$. First, we show how to compute DA (a $d \times n$-dimensional matrix). Clearly we can use $O(1)$ rounds and $O(d)$ total space to generate the entries on the diagonal of D. Note that we can easily compute each element $(DA)_{ij} = D_{ii} A_{ij}$. This is a total of nd computations. So we can do it on $(nd)^{1-\epsilon}$ machines with $(nd)^\epsilon$ local space per machine in $O(1)$ rounds.

Next, we compute $H(DA)$ (a $d \times n$-dimensional matrix). Note that H is just the d-dimensional DFT. We can then utilize the d-dimensional MPC FFT algorithm [Hajiaghayi et al., 2021] and apply it to each column of DA. For any $\epsilon \in (0, 1)$, this requires $O\left(d^\epsilon\right)$ machines, $O\left(d^{1-\epsilon}\right)$ memory, and $O(1/\varepsilon)$ rounds. Thus, we can compute all columns of DA in the MPC model using $O((nd)^\varepsilon)$ space per machine and $O((nd)^{1-\varepsilon})$ machines in $O(1/\varepsilon)$ rounds.

Finally, we compute $P(HDA)$ (a $k \times n$-dimensional matrix). Just as in Ailon and Chazelle, we see that the number of nonzero values, or $|P|$, is $\sim$ Binom(dk, q), which means:

$$\mathrm{E}[|P|] = nkq = O\left(d\xi^{-2} \log^3(n)/d\right) = O(\log^3(n)/\xi^2)$$

And then by the Markov inequality, we have $|P| = O(\log^3(n)/\xi^2)$ with probability at least 0.99. This means we only have $O(\xi^{-2} \log^3 n)$ values in P with probability at least 0.99. For each value in P, we multiply it by one row of HDA of length n for a total of $O(n\xi^{-2} \log^3 n)$ computations.

Beyond this, some of these values need to be added to find values in $PHDA$, but that will not exceed $O(n\xi^{-2}\log^3 n)$. Thus, these computations can be done in $O(1)$ rounds as long as the total space is at least $\Omega(n\xi^{-2}\log^3 n)$. Finally, there are at most d additions required to define a single entry, then we can divide this into $d^{1-\epsilon}$ sets of additions of size d^ϵ. We can then pack all these sets of computations of size at most d^ϵ into machines with memory $O(d^\epsilon)$. This requires at most $O(d^{1-\epsilon})$ machines. Then we perform the computations. To find the values for entries that require n computations, this will require $O(1/\epsilon)$ rounds. That completes the computation. □

Algorithm 19 FJLT in MPC

Input: A, a matrix of n d-dimensional vectors, and a parameter $\xi \in (0, 0.5)$
Output: $\phi(A)$
for $i \in [d]$ in parallel **do**
  Let $D_{i,i}$ be 1 or -1 with probabilities 1/2
end for
for $i \in [d], j \in [n]$ in parallel **do**
  $(DA)_{i,j} \leftarrow D_{i,i}A_{i,j}$
end for
Let $H(DA) \leftarrow$ FFT(DA, ε)
  //MPC algorithm for fast Fourier transform [Hajiaghayi et al., 2021]. Each machine holds $O((nd)^\varepsilon)$ space.
Let M be the set of multiplications of nonzero entries in P and entries in HDA
for Assign M to $(nd)^{1-\epsilon}\xi^{-2}\log^3 n/d$ machines in parallel **do**
  Compute multiplication $m \in M$ locally.
end for
Let $\mathcal{A}$ be the set of additions of m outputs in $P(HDA)$
while $\mathcal{A}$ is nonempty **do**
  for Pack large contiguous chunks of $a \in \mathcal{A}$ into $(nd)^\epsilon$ per machine in parallel **do**
    Compute a and simplify $\mathcal{A}$
  end for
end while
Store results of $\mathcal{A}$ in $\phi(A) = P(HDA)$

Part III

Adaptive Massively Parallel Graph Algorithms

Chapter 15: Adaptive Massively Parallel Constant-round Tree Contraction

15.1 Introduction

In this paper, we study and extend Miller and Reif's fundamental FOCS'85 [Miller and Reif, 1985, Miller and Reif, 1991, Miller and Reif, 1989] $O(\log n)$-round parallel *tree contraction* method. Tree contraction is a process involving iterated contraction on graph components for efficient computation of problems on trees (see Section 15.1.2). Their work leverages PRAM, a model of computation in which a large number of processors operate synchronously under a single clock and are able to randomly access a large shared memory. In PRAM, tree contractions require n processors. Though the initial study of tree contractions was in the CRCW (concurrent read from and write to shared memory) PRAM model, this was later extended to the stricter EREW (exclusive read from and write to shared memory) PRAM model [Dekel et al., 1986] as well, and then to work-optimal parallel algorithms with $O(n/\log n)$ processors [Gazit et al., 1988]. Since then, a number of additional works have also built on top of Miller and Reif's tree contraction algorithm [Acar et al., 2004, Cole and Vishkin, 1988, Gibbons and Rytter, 1989]. Tree-based computations have a breadth of applications, including natural graph problems like matching and bisection on trees, as well as problems that can be formulated on tree-like structures including expression simplification.

The tree contraction method in particular is an extremely broad technique that can be ap-

plied to many problems on trees. Miller and Reif [Miller and Reif, 1989] initially motivated their work by showing it can be used to evaluate arithmetic expressions. They additionally studied a number of other applications [Miller and Reif, 1991], using tree contractions to construct the first polylogarithmic round algorithm for tree isomorphism and maximal subtree isomorphism of unbounded degrees, compute the 3-connected components of a graph, find planar embeddings of graphs, and compute list-rankings. An incredible amount of research has been conducted to further extend the use of tree contractions for online evaluation of arithmetic circuits [Miller et al., 1988], finding planar graph separators [Gazit and Miller, 1987], approximating treewidth [Bodlaender et al., 2016], and much more [Atallah et al., 1989, Goodrich and Kosaraju, 1996, Grohe and Verbitsky, 2006, Jez and Lohrey, 2016, Miller and Ramachandran, 1987, Papadopoulos et al., 2015]. This work extends classic tree contractions to the adaptive massively parallel setting.

The importance of large-scale data processing has spurred a large interest in the study of massively parallel computing in recent years. Notably, the *Massively Parallel Computation* (MPC) model has been studied extensively in the theory community for a range of applications [Ahn and Guha, 2015b, Andoni et al., 2014, Andoni et al., 2018, Andoni et al., 2019, Assadi et al., 2019a, Assadi et al., 2019b, Assadi et al., 2019c, Bateni et al., 2017, Bateni et al., 2018b, Behnezhad et al., 2019e, Behnezhad et al., 2019d, Behnezhad et al., 2019b, Boroujeni et al., 2018, Czumaj et al., 2018, Ghaffari et al., 2018, Hajiaghayi and Knittel, 2020, Harvey et al., 2018, Lacki et al., 2020, Nanongkai and Scquizzato, 2019, Roughgarden et al., 2016, Yaroslavtsev and Vadapalli, 2018], many with a particular focus on graph problems. MPC is famous for being an abstraction of MapReduce [Karloff et al., 2010], a popular and practical programming framework that has influenced other parallel frameworks including Spark [Zaharia et al., 2016], Hadoop [White, 2009], and Flume [Chambers et al., 2010]. At a high level, in MPC, data is distributed across a

range of low-memory machines which execute local computations in rounds. At the end of each round, machines are allowed to communicate using messages that do not exceed their local space constraints. In the most challenging space-constrained version of MPC, we restrict machines to $O(n^\epsilon)$ local space for a constant $0 < \epsilon < 1$ and $\widetilde{O}(n+m)$ total space (for graphs with m edges, or just $\widetilde{O}(n)$ otherwise).

The computation bottleneck in practical implementations of massively parallel algorithms is often the amount of communication. Thus, work in MPC often focuses on *round complexity*, or the number of rounds, which should be $O(\log n)$ at a baseline. More ambitious research often strives for sublogarithmic or even constant round complexity, though this often requires very careful methods. Among others, a specific family of graph problems known as *Locally Checkable Labeling* (LCL) problems – which includes vertex coloring, edge coloring, maximal independent set, and maximal matching to name a few – admit highly efficient MPC algorithms, and have been heavily studied during recent years [Behnezhad et al., 2019e, Assadi et al., 2019b, Assadi et al., 2019a, Behnezhad et al., 2019a, Ghaffari et al., 2020, Ghaffari and Uitto, 2019, Czumaj et al., 2018]. Another consists of DP problems on sequences including edit distance [Boroujeni et al., 2018] and longest common subsequence [Hajiaghayi et al., 2019], as well as pattern matching [Hajiaghayi et al., 2021]. The round complexity of aforementioned MPC algorithms can be interpreted as the parallelization limit of the corresponding problems.

While MPC is generally an extremely efficient model, it is theoretically limited by the widely believed 1-vs-2Cycle conjecture [Ghaffari et al., 2019a], which poses that distinguishing between a graph that is a single n-cycle and a graph that is two $n/2$-cycles requires $\Omega(\log n)$ rounds in the low-memory MPC model. This has been shown to imply lower bounds on MPC round complexity for a number of other problems, including connectivity [Behnezhad et al.,

2019d], matching [Ghaffari et al., 2019a,Nanongkai and Scquizzato, 2019], clustering [Yaroslavtsev and Vadapalli, 2018], and more [Andoni et al., 2019,Ghaffari et al., 2019a,Lacki et al., 2020]. To combat these conjectured bounds, Behnezhad et al. [Behnezhad et al., 2019c] developed a stronger and practically-motivated extension of MPC, called *Adaptive Massively Parallel Computing* (AMPC). AMPC was inspired by two results showing that adding distributed hash tables to the MPC model yields more efficient algorithms for finding connected components [Kiveris et al., 2014] and creating hierarchical clusterings [Bateni et al., 2017]. AMPC models exactly this: it builds on top of MPC by allowing in-round access to a distributed read-only hash table of size $O(n+m)$. See Section 15.1.1 for a formal definition.

In their foundational work, Behnezhad et al. [Behnezhad et al., 2019c] design AMPC algorithms that outperform the MPC state-of-the-art on a number of problems. This includes solving minimum spanning tree and 2-edge connectivity in $\log\log_{m/n}(n)$ AMPC rounds (outperforming $O(\log n)$ and $O(\log D \log\log_{m/n} n)$ MPC rounds respectively), and solving maximal independent set, 2-Cycle, and forest connectivity in $O(1)$ AMPC rounds (outperforming $\widetilde{O}(\sqrt{\log n})$, $O(\log n)$, and $O(\log D \log\log_{m/n} n)$ MPC rounds respectively). Perhaps most notably, however, they proved that the 1-vs-2Cycle conjecture does not apply to AMPC by finding an algorithm to solve connectivity in $O(\log\log_{m/n} n)$ rounds. This was later improved to be $O(1/\epsilon)$ by Behnezhad et al. [Behnezhad et al., 2020], who additionally found improved algorithms for AMPC minimum spanning forest and maximum matching. Charikar, Ma, and Tan [Charikar et al., 2020] recently show that connectivity in the AMPC model requires $\Omega(1/\epsilon)$ rounds unconditionally, and thus the connectivity result of Behnezhad et al. [Behnezhad et al., 2020] is indeed tight. In a subsequent work, Behnezhad [Behnezhad, 2021] shows an $O(1/\epsilon)$-round algorithm for the maximal matching problem in AMPC.

A notable drawback of the current work in AMPC is that there is no generalized framework for solving multiple problems of a certain class. Such methods are important for providing a deeper understanding of how the strength of AMPC can be leveraged to beat MPC in general problems, and often leads to solutions for entirely different problems. Studying Miller and Reif [Miller and Reif, 1989]'s tree contraction algorithm in the context of AMPC provides exactly this benefit. We get a generalized technique for solving problems on trees, which can be extended to a range of applications.

Recently, Bateni et al. [Bateni et al., 2018a] introduced a generalized method for solving "polylog-expressible" and "linear-expressible" dynamic programs on trees in the MPC model. This was heavily inspired by tree contractions, and also is a significant inspiration to our work. Specifically, their method solves minimum bisection, minimum k-spanning tree, maximum weighted matching, and a large number of other problems in $O(\log n)$ rounds. We extend these methods, as well as the original tree contraction methods, to the AMPC model to create more general techniques that solve many problems in $O_\epsilon(1)$ rounds.

15.1.1 The AMPC Model

The AMPC model, introduced by Behnezhad et. al [Behnezhad et al., 2019c], is an extension of the standard MPC model with additional access to a *distributed hash table*. In MPC, data is initially distributed across machines and then computation proceeds in rounds where machines execute local computations and then are able to share small messages with each other before the next round of computation. A distributed hash table stores a collection of key-value pairs which are accessible from every machine, and it is required that both key and value have a

constant size. Each machine can adaptively query a bounded sequence of keys from a centralized distributed hash table during each round, and write a bounded number of key-value pairs to a distinct distributed hash table which is accessible to all machines in the next round. The distributed hash tables can also be utilized as the means of communication between the machines, which is implicitly handled in the MPC model, as well as a place to store the initial input of the problem. It is straight-forward to see how every MPC algorithm can be implemented within the same guarantees for the round-complexity and memory requirements in the AMPC model.

Definition 155. *Consider a given graph on n vertices and m edges. In the **AMPC model**, there are $\mathcal{P}$ machines each with **sublinear** local space $\mathsf{M} = O(n^\epsilon)$ for some constant $0 < \epsilon < 1$, and the total memory of machines is bounded by $\widetilde{O}(n+m)$. In addition, there exist a collection of distributed hash tables $\mathcal{H}_0, \mathcal{H}_1, \mathcal{H}_2, \ldots$, where $\mathcal{H}_0$ contains the initial input.*

*The process consists of several rounds. During round i, each machine is allowed to make at most $O(\mathsf{M})$ read queries from $\mathcal{H}_{i-1}$ and to write at most $O(\mathsf{M})$ key-value pairs to $\mathcal{H}_i$. Meanwhile, the machines are allowed to perform an arbitrary amount of computation locally. Therefore, it is possible for machines to decide what to query next after observing the result of previous queries. In this sense, the queries in this model are **adaptive**.*

15.1.2 Our Contributions

The goal of this paper is to present a framework for solving various problems on trees with constant-round algorithms in AMPC. This is a general strategy, where we intelligently shrink the tree iteratively via a *decomposition* and *contraction* process. Specifically, we follow Miller and Reif's [Miller and Reif, 1989] two-stage process, where we first *compress* each connected

component in our decomposition, and then *rake* the leaves by contracting all leaves of the same parent together. We repeat until we are left with a single vertex, from which we can extract a solution. To retrieve the solution when the output corresponds to many vertices in the tree (i.e., maximum matching instead of maximum matching value), we can undo the contractions in reverse order and populate the output as we gradually reconstruct the original tree.

The decomposition strategy must be constructed very carefully such that we do not lose too much information to solve the original problem and each connected component must fit on a single machine with $O(n^\epsilon)$ local memory. To compress, we require oracle access to a black-box function, a *connected contracting function*, which can efficiently contract a connected component into a vertex while also retaining enough information to solve the original problem. To rake leaves, we require oracle access to another block-box function, a *sibling contracting function*, which executes the same thing but on a set of leaves that share a parent. These two black-box functions are problem specific (e.g., we need a different set of functions for maximum matching and maximum independent set). In this paper,

we only require contracting functions to accept n^ϵ vertices as the input subgraphs, and we always run these black-box functions locally on a single machine. Thus, we can compress any arbitrary collection of disjoint components of size at most n^ϵ in $O(1)$ AMPC rounds. See Section 15.2.1 for formal definitions.

This general strategy actually works on a special class of structures, called *degree-weighted trees* (defined in §15.2). Effectively, these are trees $T = (V, E, W)$ with a multi-dimensional weight function where $W(v) \in \{0,1\}^{\widetilde{O}(\deg(v))}$ stores a vector of bits proportional in size to the degree of the vertex $v \in V$. When we use our contracting functions, we use W to store data about the set of vertices we are contracting. This is what allows our algorithms to retain enough

information to construct a solution to the entire tree T when we contract sets of vertices. Note that the degree of the surviving vertex after contraction could be much smaller than the total degree of the original set of vertices.

Our first algorithm works on trees with bounded degree, more precisely, trees with maximum degree at most n^ϵ. The reason this is easier is because when an internal connected component is contracted, we often need to encode the output of the subproblem at the root (e.g., the maximum weighted matching on the rooted subtree) in terms of the children of this component post-contraction. In high degree graphs, it may have many children after being contracted, and therefore require a large encoding (i.e., one larger than $O(n^\epsilon)$) and thus not fit on one machine.

In this algorithm, we find that if the degree is bounded by n^ϵ and we compress sufficiently small components, then the algorithm works out much more smoothly. The underlying technique that allows us to contract the tree into a single vertex in $O(1/\epsilon)$ iterations is a decomposition of vertices based on their preorder numbering. The surprising fact is that each group in this decomposition contains at most one non-leaf vertex after contracting connected components. Thus, an additional single rake stage is sufficient to collapse any tree with n vertices to a tree with at most $n^{1-\epsilon}$ vertices in a single iteration. However, we need $O(1/\epsilon)$ AMPC rounds at the beginning of each iteration to find the decomposition associated with the resulting tree after contractions performed in the previous iteration. This becomes $O(1/\epsilon^2)$ AMPC rounds across all iterations. See Section 15.3.1 for the proofs and more details.

This is a nice independent result, proving a slightly more efficient $O(1/\epsilon^2)$-round algorithm on degree bounded trees. Additionally, many problems on larger degree trees can be represented by lower degree graphs. For example, both the original Miller and Reif [Miller and Reif, 1985] tree contraction and the Betani et al. [Bateni et al., 2018a] framework consider only problems in

which we can replace each high degree vertex by a balanced binary tree, reducing the tree-based computation on general trees to a slightly different computation on binary trees. Equally notably, it is an important subroutine in our main algorithm.

Theorem 156. *Consider a degree-weighted tree $T = (V, E, W)$ and a problem P. Given a connected contracting function on T with respect to P, one can compute $P(T)$ in $O(1/\epsilon^2)$ AMPC rounds with $O(n^\epsilon)$ memory per machine and $\widetilde{O}(n)$ total memory if $\deg(v) \leq n^\epsilon$ for every vertex $v \in V$.*

Remark 157. *It may be tempting to suggest that in most natural problems the input tree can be transformed into a tree with degree bounded by n^ϵ. However, we briefly pose the* MedianParent *problem, where leaves are given values and parents are defined recursively as the median of their children. By transforming the tree to make it degree bounded, we lose necessary information to find the median value among the children of a high degree vertex.*

Next, we move onto our main result: a generalized tree contraction algorithm that works on any input tree with arbitrary structure. Building on top of Theorem 156, we can create a natural extension of tree contractions. Recall that the black-box contracting functions encode the data associated with a contracted vertex in terms of its children post-contraction. Thus, allowing high degree vertices introduces difficulties working with contracting functions. In particular, it is not possible to store the weight vector $W(v)$ of a high degree vertex v inside the local memory of a single machine. The power of this algorithm is its ability to implement COMPRESS and RAKE for n^ϵ-tree-contractions in $O(1/\epsilon^3)$ rounds.

The most significant novelty of our main algorithm is the handling of high degree vertices. To do this, we first handle all maximal connected components of low degree vertices using the

algorithm from Theorem 156 as a black-box. This compresses each such component into one vertex without needing to handle high degree vertices. By contracting these components, we obtain a special tree called *Big-Small-tree* (defined formally in §15.3.2) which exhibits nice structural properties. Since the low degree components are maximal, the degree of each vertex in every other layer is at least n^ϵ, implying that a large fraction of the vertices in a Big-Small-tree must be leaves. Hence, after a single rake stage, the number of high degree vertices drops by a factor of n^ϵ.

In order to rake the leaves of high degree vertices, we have to carefully apply our sibling contracting functions in a way that can be implemented efficiently in AMPC. Unlike Theorem 156 in which having access to a connected contracting function is sufficient, here we also require a sibling contracting function. Consider a star tree with its center at the root. Without a sibling contracting function, we are able to contract at most $O(n^\epsilon)$ vertices in each round since the components we pass to the contracting functions must be disjoint. But having access to a sibling contracting function, we can rake up to $O(n)$ leaf children of a high degree vertex in $O(1/\epsilon)$ rounds. For more details about the algorithm and proofs see Section 15.3.2.

Theorem 158. *Consider a degree-weighted tree $T = (V, E, W)$ and a problem P. Given a connected contracting function and a sibling contracting function on T with respect to P, one can compute $P(T)$ in $O(1/\epsilon^3)$ AMPC rounds with $O(n^\epsilon)$ memory per machine and $\widetilde{O}(n)$ total memory.*

Theorem 156 and Theorem 158 give us general tools that have the power to create efficient AMPC algorithms for any problem that admits a connected contracting function and a sibling contracting function. Intuitively, they reduce constant-round parallel algorithms for a specific

problem on trees to designing black-box contracting functions that are sequential. We should be careful in designing contracting functions to make sure that the amount of data stored in the surviving vertex does not asymptotically exceed its degree in the contracted tree. Also note that a connected contracting function works with unknown values that depend on the result of other components.

Satisfying these conditions is a factor that limits the extent of problems that can be solved using our framework. For example, the framework of Bateni et. al [Bateni et al., 2018a] works on a wider range of problems on trees since their algorithm, roughly speaking, tolerates exponential growth of weight vectors using a careful decomposition of tree. Indeed, they achieve these benefits at the cost of an inherent requirement for at least $O(\log n)$ rounds due to the divide-and-conquer nature of their algorithm. However, their framework comes short on addressing problems such as MedianParent (defined in Remark 157) that are not reducible to binary trees. Nonetheless, we show several techniques for designing contracting functions that satisfy these conditions, in particular:

1. In Section 15.3.3, we prove a general approach for designing a connected contracting function and a sibling contracting function given a PRAM algorithm based on the original Miller and Reif [Miller and Reif, 1985] tree contraction. We do this by observing that in almost every conventional application of Miller and Reif's framework, the length of data stored at each vertex remains constant throughout the algorithm.

2. Storing a minimal tree representation of a connected component contracted into v in the weight vector $W(v)$ enables us to simplify a recursive function defined on the subtree rooted at v in terms of yet-unknown values of its children, while keeping the length of $W(v)$

asymptotically proportional to $deg(v)$. For instance, our maximum weighted matching algorithm (See Section 4.1 of the full version for more details) uses this approach.

Ultimately, this is a highly efficient generalization of the powerful tree contraction algorithm. To illustrate the versatility of our framework, we show that it gives us efficient AMPC algorithms for many important applications of frameworks such as Miller and Reif [Miller and Reif, 1989]'s and Bateni et al. [Bateni et al., 2018a]'s by constructing sequential black-box contracting functions. In doing so, we utilize a diverse set of techniques, including the ones mentioned above, that are of independent interest and can be applied to a broad range of problems on trees. The proof Theorem 159 and more details about each application can be found in the full version.

Theorem 159. *Algorithms 20 and 21 can solve, among other applications, dynamic expression evaluation, tree isomorphism testing, maximal matching, and maximal independent set in $O(1/\epsilon^2)$ AMPC rounds, and maximum weighted matching and maximum weighted independent set in $O(1/\epsilon^3)$ AMPC rounds. All algorithms use $O(n^\epsilon)$ memory per machine and $\widetilde{O}(n)$ total memory.*

15.1.3 Paper Outline

The work presented in this paper is a constant-round generalized technique for solving a large number of graph theoretic problems on trees in the AMPC model. In Section 15.2, we go over some notable definitions and conventions we will be using throughout the paper. This includes the introduction of a generalized weighted tree, a formalization of the general tree contraction process, the definition of contracting functions, and a discussion of a tree decomposition

method we call the *preorder decomposition*. In the Section 15.3, we go over our main results, algorithms, and proofs. The first result (§15.3.1) is an algorithm for executing a tree contraction-like process which solves the same problems on trees of bounded maximum degree. The second result (§15.3.2) utilizes the first result as well as additional novel techniques to implement generalized tree contractions. We additionally show (§15.3.3) that our algorithms can also implement Miller and Reif's standard notion of tree contractions, and (§15.3.4) we show how to efficiently reconstruct a solution on the entire graph by reversing the tree contracting process.

15.2 Preliminaries

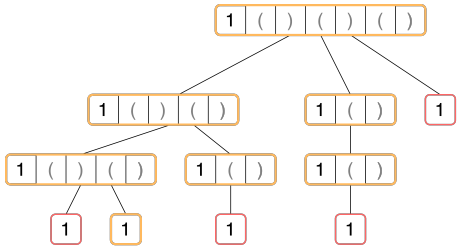
In this work, we are interested in solving problems on trees $T = (V, E)$ where $|V| = n$. Our algorithms iteratively transform T by contracting components in an intelligent way that: (1) components can be stored on a single machine, (2) the number of iterations required to contract T to a single vertex is small, and (3) at each step of the process, we still have enough information to solve the initial problem on T.

To achieve (3), we must retain some information about an original component after we contract it. For instance, consider computing all maximal subtree sizes. For a connected component S with $r = \text{lca}(S)$[1], the contracted vertex v_S of S might encode $|S|$ and a list of its leaves (when viewing S as a tree itself). It is not difficult to see that this would be sufficient knowledge to compute all maximal subtree sizes for the rest of the vertices in T without considering all individual vertices in S. Data such as this is encoded as a multi-dimensional weight function which maps vertices to binary vectors. We will specifically consider trees where the dimensionality of the weight function is bounded by the degree of the vertex.

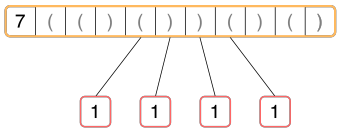
[1] lca is the least common ancestor function.

We note that in this paper, when we refer to the degree of a vertex in a rooted tree, we ignore parents. Therefore, $\deg(v)$ is the number of children a vertex has.

Definition 160. *A **degree-weighted tree** is a tree $T = (V, E, W)$ with vertex set V, edge set E, and vertex weight vector function W such that for all $v \in V$, $W(v) \in \{0,1\}^{\widetilde{O}(\deg(v))}$.*[2]



(a) Each vertex in the degree weighted tree T stores the size of its subtree, which is 1 initially and the structure of the subtree between each vertex and its children using parenthesis notation which is simply a star for every vertex at the beginning. In the parenthesis notation, we traverse the tree according to the preorder numbering and put an '(' whenever we go down from a parent to a child, and a ')' whenever we go up from a child to a parent.



(b) In the contracted degree weighted tree T', the structure of the yellow subgraph is recorded in the weight vector of the root, a tree with 4 leaves (equal to the degree of root) which is not a star. In addition, the size of the contracted subgraph is stored in the weight vector of the root.

Figure 15.1: A degree weighted tree $T = (V, E, W)$ with $|V| = 11$. In Subfigure 15.1a, we have a degree weighted tree with $|W_v| \leq 4deg(v)$. We contract the subgraph with 7 vertices depicted by yellow in Subfigure 15.1a using a connected contracting function (Defined in Definition 162). The resulting degree weighted tree T' is depicted in Subfigure 15.1b. Note that the length of weight vectors in proportional to the degree of each vertex even after the contraction.

Notationally, we let $w(v) = \dim(W(v)) = \widetilde{O}(\deg(v))$ be the length of the weight vectors. Additionally, note that a tree $T = (V, E)$ is a degree-weighted tree where $W(v) = \emptyset$ for all $v \in V$.

In order to implement our algorithm, we also require specific *contracting functions* whose properties allow us to achieve the desired result (§15.2.1). In addition, we will introduce a specific tree decomposition method, called a *preorder decomposition*, that we will efficiently implement

[2] $\widetilde{O}(f(n)) = O(f(n)\log n)$.

and leverage in our final algorithms (§15.2.2).

15.2.1 Tree Contractions and Contracting Functions

Our algorithms provide highly efficient generalizations to Miller and Reif's [Miller and Reif, 1989] tree contraction algorithms. At a high level, their framework provides the means to compute a global property with respect to a given tree in $O(\log n)$ *phases*. In each *phase*, there are two *stages*:

- COMPRESS stage: Contract around half of the vertices with degree 1 into their parent.
- RAKE stage: Contract all the leaves (vertices with degree 0) into their parent.

Repeated application of COMPRESS and RAKE alternatively results in a tree which has only one vertex. Intuitively, the COMPRESS stage aims to shorten the long *chains*, maximal connected sequences of vertices whose degree is equal to 1, and the RAKE stage cleans up the leaves. Both stages are necessary in order to guarantee that $O(\log n)$ phases are enough to end up with a single remaining vertex [Miller and Reif, 1989].

In the original variant, every odd-indexed vertex of each *chain* is contracted in a COMPRESS stage. In some randomized variants, each vertex is selected with probability $1/2$ independently, and an independent set of the selected vertices is contracted. In such variants, contracting two consecutive vertices in a chain is avoided in order to efficiently implement the tree contraction in the PRAM model. However, this restriction is not imposed in the AMPC model, and hence we consider a more *relaxed variant* of the COMPRESS stage where each maximal chain is contracted into a single vertex.

We introduce a more generalized version of tree contraction called α-*tree-contractions*. Here, the RAKE stage is the same as before, but in the COMPRESS stage, every maximal subgraph containing only vertices with degree less than α is contracted into a single vertex.

Definition 161. *In an α-tree-contraction of a tree $T = (V, E)$, we repeat two stages in a number of phases until the whole tree is contracted into a single vertex:*

- COMPRESS *stage: Contract every maximal connected component S containing only vertices with degree less than α, i.e., $deg(v) < \alpha \ \ \forall v \in S$, into a single vertex S'.*
- RAKE *stage: Contract all the leaves into their parent.*

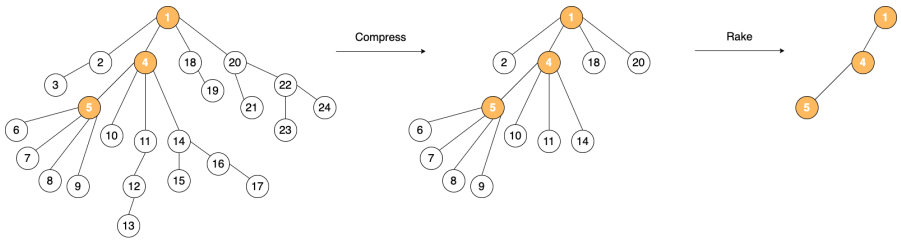


Figure 15.2: An example phase in α-tree-contraction for $\alpha = 4$. In the leftmost tree the initial tree is depicted, and the vertices are numbered from 1 to n in the preorder ordering. In the middle tree, we performed a COMPRESS stage to get a tree with 12 vertices. Next, we RAKE the leaves to end up with a tree with 3 vertices depicted on the right.

Notice that the relaxed variant of Miller and Reif's COMPRESS stage is the special case when $\alpha = 2$. Our goal will be to implement efficient α-tree-contractions where $\alpha = n^{\epsilon}$.

In order to implement COMPRESS and RAKE, we need fundamental tools for contracting a single set of vertices into each other. We call these *contracting functions*. In the COMPRESS stage, we must contract connected components. In the RAKE stage, we must contract leaves with the same parent into a single vertex. These functions run locally on small sets of vertices.

Definition 162. *Let P be some problem on degree-weighted trees such that for some degree-weighted tree T, $P(T)$ is the solution to the problem on T. A* ***contracting function*** *on T with*

respect to P is a function f that replaces a set of vertices in T with a single vertex and incident edges to form a degree-weighted tree T' such that $P(T) = P(T')$[3]. There are two types:

1. *f is a **connected contracting function** if f contracts[4] connected components into a single vertex of T.*

2. *f is a **sibling contracting function** if f is defined on sets of leaf siblings (i.e., leaves that share a parent p) of T, and the new vertex is a leaf child of p.*

Since the output of the contracting function is a degree-weighted tree, it implicitly must create a weight $W(v)$ for any newly contracted vertex v.

15.2.2 Preorder Decomposition

A *preorder decomposition* (formally defined shortly) is a strategy for decomposing trees into a disjoint union of (possibly not connected) vertex groups. In this paper, we will show that the preorder decomposition exhibits a number of nice properties (see §15.3) that will be necessary for our tree contraction algorithms. Ultimately, we wish to find a decomposition of vertices $V_1, V_2, \ldots, V_k \subseteq V$ of a given tree $T = (V, E)$ ($\cup_{i=1}^{k} V_i = V$ and $V_i \cap V_j = \phi \quad \forall i, j : i \neq j$) so that for all $i \in [k]$, after contracting each connected component contained in the same vertex group, the maximum degree is bounded by some given λ. Obviously, this won't be generally possible (i.e., consider a large star), but we will show that this holds when the maximum degree of the input tree is bounded as well.

[3]With some nuance, it depends on the format of the problem. For instance, when computing the value of the maximum independent set, the single values $P(T)$ and $P(T')$ should be the same. When computing the maximum independent set itself, uncontracted vertices must have the same membership in the set, and contracted vertices represent their roots.

[4]Consider a connected component S with a set of external neighbors $N(S) = \{v \in V \setminus S : \exists u \in S (v, u) \in E\}$. Then contracting S means replacing S with a single vertex with neighborhood $N(S)$.

The preorder decomposition is depicted in Figure 15.3a. Number the vertices by their index in the preorder traversal of tree T, i.e., vertices are numbered $1, 2, \ldots, n$ where vertex i is the i-th vertex that is visited in the preorder traversal starting from vertex 1 as root. In a preorder decomposition of T, each group V_i consists of a consecutive set of vertices in the preorder numbering of the vertices. More precisely, let l_i denote the index of the vertex $v \in V_i$ with the largest index, and assume $l_0 = 0$ for consistency. In a preorder decomposition, group V_i consists of vertices $l_{i-1} + 1, l_{i-1} + 2, \ldots, l_i$.

Definition 163. *Given a tree $T = (V, E)$, a* "preorder decomposition" *$V_1, V_2, \ldots, V_k$ of T is defined by a vector $l \in \mathbb{Z}^{k+1}$, such that $0 = l_0 < l_1 < \ldots < l_k = n$, as $V_i = \{l_{i-1} + 1, l_{i-1} + 2, \ldots, l_i\}$ $\forall i \in [k]$. See Subfigure 15.3a for an example.*

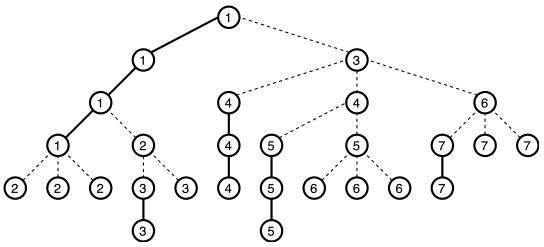
Assume we want each V_i in our preorder decomposition to satisfy $\sum_{v \in V_i} \deg(v) \leq \lambda$ for some λ. As long as $\deg(v) \leq \lambda$ for all $v \in V$, we can greedily construct components $V_1, \ldots, V_k$ according to the preorder traversal, only stopping when the next vertex violates the constraint. Since $\sum_{v \in V} \deg(v) \leq n$, it is not hard to see that this will result in $O(n/\lambda)$ groups that satisfy the degree sum constraint.

Observation 164. *Consider a given tree $T = (V, E)$. For any parameter λ such that $\deg(v) \leq \lambda$ for all $v \in V$, there is a preorder decomposition $V_1, V_2, \ldots, V_k$ such that $\forall i \in [k]$, $\sum_{v \in V_i} \deg(v) \leq \lambda$, and $k = O(n/\lambda)$.*

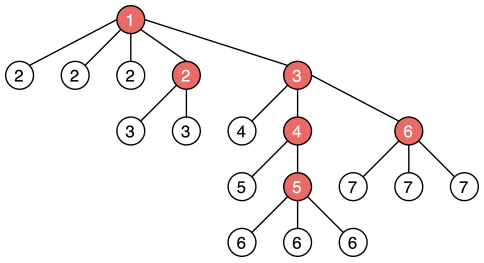
The *dependency tree* $T' = (V', E')$, as seen in Figure 15.3b of a decomposition is useful notion for understanding the structure of the resulting graph. In T', vertices represent connected components within groups, and there is an edge between vertices if one contains a vertex that is

a parent of a vertex in the other. This represents our contraction process and will be useful for bounding the size of the graph after each step.

Definition 165. *Given a tree $T = (V, E)$ and a decomposition of vertices $V_1, V_2, \ldots, V_k$, the **dependency tree** $T' = (V', E')$ of T under this decomposition is constructed by contracting each connected component $C_{i,j}$ for all $j \in [c_i]$ in each group V_i. We call a component contracted to a leaf in T' an **independent** component, and a component contracted to a non-leaf vertex in T' a **dependent** component.*



(a) An example preorder decomposition of T into $V_1, V_2, \ldots, V_7$ with $\lambda = 8$. Edges within any F_i are depicted bold, and edges belonging to no F_i are depicted dashed.



(b) Dependency tree T', created by contracting connected components of every F_i. Each red vertex represents a dependent component, and each white vertex represents an independent component.

Figure 15.3: In Subfigure (a), a preorder decomposition of a given tree T is demonstrated. Based on this preorder decomposition, we define a dependency tree T' so that each connected component S in each forest F_i is contracted into a single vertex S'. This dependency tree T' is demonstrated in Subfigure (b). It is easy to observe that the contracted components are maximal components which are connected using bold edges in T, and each edge in T' corresponds to a dashed edge in T.

15.3 Constant-round Tree Contractions in AMPC

The main results of this paper are two new algorithms. The first algorithm applies α-tree-contraction-*like* methods in order to solve problems on trees where the degrees are bounded by n^{ϵ}. Though this algorithm is similar in inspiration to the notion of tree contractions, it is not a

true α-tree-contraction method.

Theorem 156. *Consider a degree-weighted tree $T = (V, E, W)$ and a problem P. Given a connected contracting function on T with respect to P, one can compute $P(T)$ in $O(1/\epsilon^2)$ AMPC rounds with $O(n^\epsilon)$ memory per machine and $\widetilde{O}(n)$ total memory if $\deg(v) \leq n^\epsilon$ for every vertex $v \in V$.*

This algorithm provides us with two benefits: (1) it is a standalone result that is quite powerful in its own right and (2) it is leveraged in our main algorithm for Theorem 158. The only differences between this result and our main result for generalized tree contractions is that we require $\deg(v) \leq n^\epsilon$, but it runs in $O(1/\epsilon^2)$ rounds, as opposed to $O(1/\epsilon^3)$ rounds. Thus, if the input tree has degree bounded by n^ϵ, then clearly the precondition is satisfied. Additionally, if the tree can be decomposed into a tree with bounded degree such that we can still solve the problem on the decomposed tree, this result applies as well.

Our general results are quite similar, with a slightly worse round complexity, but with the ability to solve the problem on all trees. Notably, it is a true α-tree-contraction algorithm.

Theorem 158. *Consider a degree-weighted tree $T = (V, E, W)$ and a problem P. Given a connected contracting function and a sibling contracting function on T with respect to P, one can compute $P(T)$ in $O(1/\epsilon^3)$ AMPC rounds with $O(n^\epsilon)$ memory per machine and $\widetilde{O}(n)$ total memory.*

In this section, we introduce both algorithms and prove both theorems.

15.3.1 Contractions on Degree-Bounded Trees

We now provide an $O(1/\epsilon^2)$-round AMPC algorithm with local space $O(n^\epsilon)$ for solving any problem P on a degree-weighted tree $T = (V, E, W)$ with bounded degree $\deg(v) \leq n^\epsilon$ for all $v \in V$ given a *connected contracting function* for P. The method, which we call BoundedTreeContract, can be seen in Algorithm 20.

Much like an α-tree-contraction algorithm, it can be divided into a COMPRESS and RAKE stage. In the COMPRESS stage, instead of compressing the whole maximal components that consist of low-degree vertices as required for α-tree-contractions, we partition the vertices into groups using a preorder decomposition and bounding the group size by n^ϵ. In the RAKE stage, since the degree is bounded by n^ϵ, all leaves who are children of the same vertex can fit on one machine. Thus each sibling contraction that must occur can be computed entirely locally. If we include the parent of the siblings, we can simply apply COMPRESS's connected contracting function on the children. This is why we do not need a sibling contracting function.

Let $T_0 = T$ be the input tree. For every iteration $i \in [O(1/\epsilon)]$: (1) find a preorder decomposition $V_1, \ldots V_k$ of T_{i-1}, (2) contract each connected component in the preorder decomposition, and (3) put each maximal set of leaf-siblings (i.e., leaves that share a parent) in one machine and contract them into their parent. We sometimes refer to these maximal sets of leaf-siblings by *leaf-stars*. After sufficiently many iterations, this should reduce the problem to a single vertex, and we can simply solve the problem on the vertex.

Notice that we can view the first and second steps as the COMPRESS stage except that we limit each component such that the sum of the degrees in each component is at most n^ϵ. Since the size of the vector $W(v)$ is $w(v) = \widetilde{O}(\deg(v)) = \widetilde{O}(n^\epsilon)$, we can store an entire component (in

Algorithm 20 BoundedTreeContract
(Computing the solution $P(T)$ of a problem P on degree-weighted tree T with max degree n^ϵ using connected contracting function $\mathcal{C}$)

Input Degree-weighted tree $T = (V, E, W)$ with degree bounded by n^ϵ and a connected contracting function $\mathcal{C}$.

Output he problem output $P(T)$.

$T_0 \leftarrow T$
for $i \leftarrow 1$ to $l = O(1/\epsilon)$ **do**
    Let $\mathcal{O}$ be a preordering of V_{i-1}
    Find a *preorder decomposition* $V_1, V_2, \ldots, V_k$ of T_{i-1} with
    $\lambda = n^\epsilon$ using $\mathcal{O}$
    Let $\mathcal{S}_{i-1}$ be the set of all connected components in V_i for all $i \in [k]$
    Let T'_{i-1} be the result of contracting $\mathcal{C}(S_{i-1,j})$ for all $S_{i-1,j} \in \mathcal{S}_{i-1}$
    Let $\mathcal{L}_{i-1}$ be the set of all maximal leaf-stars (containing their parent) in T'_{i-1}
    Let T_i be the result of contracting $\mathcal{C}(L_{i-1,j})$ for all $L_{i-1,j} \in \mathcal{L}_{i-1}$
end for
Return $\mathcal{C}(T_l)$

its current, compressed state) in a single machine, thus making the second step distributable. The third step can be viewed as a RAKE function which, as we stated, can be handled on one machine per contraction using the connected contracting function.

In order to get $O(1/\epsilon^2)$ rounds, we first would like to show that the number of phases is bounded by $O(1/\epsilon)$. To prove this, we show that there will be at most one non-leaf node after we contract the components in each group. In other words, the dependency tree resulting from the preorder decomposition has at most one non-leaf node per group in the decomposition. This is a necessary property of decomposing the tree based off the preorder traversal. To see why this is true, consider a connected component in a partition. If it is *not* the last connected component (i.e., it does not contain the partition's last vertex according to the preorder numbering), then after contracting, it cannot have any children.

Lemma 166. *The dependency tree $T' = (V', E')$ of a preorder decomposition $V_1, V_2, \ldots, V_k$ of tree $T = (V, E)$ contains at most 1 non-leaf vertex per group for a total of at most k non-leaf*

vertices. In other words, there are at most k dependent connected components in $\cup_{i\in[k]}F_i$.

Proof. Each group V_i induces a forest F_i on tree T, and recall that each F_i is consisted of multiple connected components $C_{i,1}, C_{i,2}, \ldots, C_{i,c_i}$, where c_i is the number of connected components of F_i. Assume w.l.o.g. component C_{i,c_i} is the component which contains vertex l_i, the vertex with the largest index in V_i. We show that every connected component in F_i except C_{i,c_i} is independent, and thus Lemma 166 statement is implied. See in Subfigure 15.3b that there is at most 1 dependent component, red vertices in T', for each group V_i. Also note that C_{i,c_i}, the only possibly dependent component in F_i, is always the last component if we sort the components based on their starting index since $l_i \in C_{i,c_i}$ and each $C_{i,j}$ contains a consecutive set of vertices.

Assume for contradiction that there exists $C_{i,j}$ for some $i \in [k], j \in [c_i - 1]$, a non-last component in group V_i, such that $C_{i,j}$ is a dependent component, or equivalently $C'_{i,j}$ is not a leaf in T'. Since $C_{i,j}$ is a dependent component, there is a vertex $v \in C_{i,j}$ which has a child outside of V_i. Let u be the first such child of v in the pre-order traversal, and thus $u \in V_j$ for some $j > i$. Consider a vertex $w \in V_i$ that comes after v in the pre-order traversal. Then, since u, and thus V_j, comes after v and V_i in the pre-order traversal, u must come after w in the pre-order traversal. Since w is between v and u in the pre-order traversal, and u is a child of v, the only option is for w to be a descendant of v. Then the path from w to v consists of w, w's parent $p(w)$, $p(w)$'s parent, and so on until we reach v. Since a parent always comes before a child in a pre-order traversal, all the intermediate vertices on the path from w to v come between w and v in the pre-order traversal, so they must all be in V_i. This means w is in $C_{i,j}$ since w is connected to v in F_i. Since any vertex after v in V_i must be in $C_{i,j}$, $C_{i,j}$ must be the last connected component, i.e., $j = c_i$. This implies that the only possibly dependent connected component of F_i is C_{i,c_i}, and all

other $C_{i,j}$'s for $j \in [c_i - 1]$ are independent. □

Lemma 166 nicely fits with our result from Observation 164 to bound the total number of phases BoundedContract requires. In addition, we can show how to implement each phase to bound the complexity of our algorithm. Note that we are assuming that our component contracting function is defined to always yield a degree-weighted tree. We only need to show that the degrees stay bounded throughout the algorithm.

Proof of Theorem 156. In each phase of this algorithm, the only modifications to the graph that occur are applications of the connected contracting functions to connected components of the tree. Since these are assumed to preserve $P(T)$ and we simply solve $P(T_l)$ for the final tree T_l, correctness of the output is obvious.

An important invariant in this algorithm is the $O(n^\epsilon)$ bound on the degree of vertices throughout the algorithm. At the beginning, we know that the degrees are bounded according as it is promised in the input. We show that this bound on the maximum degree of the tree is invariant by proving the degree of vertices are still bounded after a single contraction.

Recall that we use preorder decomposition with $\lambda = n^\epsilon$ to find the connected components we need to contract in the COMPRESS stage. According to definition, the total degree of each group in our decomposition is bounded by λ. After we contract a component S, the degree of the contracted vertex v^S never exceeds the sum of the degree of all vertices in S since every child of v^S is a child of exactly one of the vertices in S. Thus, the degree of v^S is bounded by $\lambda = n^\epsilon$. In RAKE stage, we contract a number of sibling leaves into their common parent. In this case, the degree of the parent only decreases and the bound still holds.

We now focus on round and space complexities. A preordering can be computed using the

preorder traversal algorithm from Behnezhad et al. [Behnezhad et al., 2019c], which executed in $O(1/\epsilon)$ rounds with $O(n^\epsilon)$ local space and $\widetilde{O}(n)$ total space w.h.p.[5] This completes step 1. In steps 2 and 3, the contracting functions are applied in parallel for a total of $O(1)$ rounds (based off our assumption about any given contracting functions) within the same space constraints. Thus, all phases require $O(1)$ rounds except the first, which is $O(1/\epsilon)$ rounds, and satisfy the space constraints of our theorem.

Now we must count the phases. Lemma 166 tells us that for every group, we only have one non-leaf component in the dependency graph after each step 2. In step 3, we then "RAKE" all leaves into their parents. This means that the remaining number of vertices after step 3 is equal to the number of non-leaf vertices in the dependency graph after step 2, which is $k = n^\epsilon$. Observation 164 tells us that the resulting graph size is then $O(n/n^\epsilon) = O(n^{1-\epsilon})$. Therefore, in order to get a graph where $|T_l| = 1$, we require $O(1/\epsilon)$ phases. Combining this with the complexity of each phase yields the desired result. □

15.3.2 Generalized α-Tree-Contractions

In the rest of this section we prove our main result: a generalized tree contraction algorithm, Algorithm 21. Building on top of Theorem 156, we can create a natural extension of tree contractions. Recall from §15.2 that in the COMPRESS stage, we must contract maximal connected components containing only vertices v with degree $d(v) < \alpha$. Conveniently, by Theorem 156, Algorithm 20 achieves precisely this. Therefore, to implement tree contractions, we simply need to:

1. Identify maximal connected components of low degree (Algorithm 21, line 3), which can

[5] This means with probability at least $1/poly(n)$

be done in $O(1/\epsilon)$ rounds by Behnezhad et al. [Behnezhad et al., 2020].

2. Use our previous algorithm to execute the COMPRESS stage on each component (Algorithm 21, line 5), which can be done by Algorithm 20 in $O(1/\epsilon^2)$ rounds.

3. Apply a function that can execute the RAKE stage (Algorithm 21, lines 7 through 14).

To satisfy the third step, we use a *sibling contracting function* (Definition 162), which can contract leaf-siblings of the same parent into a single leaf. Since a vertex might have up to n children, to do this in parallel, we may have to group siblings into n^ϵ-sized groups and repeatedly contract until we reach one leaf. Assuming sibling contractions are locally performed inside machines, this will then take $O(1/\epsilon)$ AMPC rounds.

Algorithm 21 TreeContract
(Computing the solution $P(T)$ of a problem P on degree-weighted tree T using a connected contracting function $\mathcal{C}$ and a sibling contracting function $\mathcal{R}$)

Input Degree-weighted tree $T = (V, E, W)$, a connected contracting function $\mathcal{C}$, and a sibling contracting function $\mathcal{R}$.
Output The problem output $P(T)$.

1: $T_0 \leftarrow T$
2: **for** $i \leftarrow 1$ to $l = O(1/\epsilon)$ **do**
3: Let $\mathcal{S}_{i-1} \leftarrow$ Connectivity$(T_{i-1} \setminus \{v \in V : deg(v) > n^\epsilon)$
4: Let $\mathcal{K}_{i-1} \leftarrow$ Components in $\mathcal{S}_{i-1,j}$ which represent a leaf in T'_{i-1}
5: Contract each $S_{i-1,j} \in \mathcal{K}_{i-1}$ into $S'_{i-1,j}$ by applying BoundedTreeContract$(S_j, \mathcal{C})$
6: Let $\mathcal{L}_{i-1}$ be the set of all maximal leaf-stars (excluding their parent) in T'_{i-1}
7: **for** $L_{i-1,0} = \{v_1, v_2, \ldots, v_k\} \in \mathcal{L}_{i-1}$ **do**
8: **for** $j \leftarrow 1$ to $1/\epsilon$ **do**
9: Split $L_{i-1,j-1}$ into $k/n^{j\epsilon}$ parts $L_{i-1,j,1}, \ldots, L_{i-1,j,k/n^{j\epsilon}}$ each of size n^ϵ
10: Contract each $L_{i-1,j,z}$ into $L'_{i-1,j,z}$ by applying $\mathcal{R}(L_{i-1,j,z})$
11: Let $L_{i-1,j} \leftarrow \{L'_{i-1,j,1}, \ldots, L'_{i-1,j,k/n^{j\epsilon}}\}$
12: **end for**
13: Contract $L_{i-1,1/\epsilon}$ by applying $\mathcal{R}(L_{i-1,1/\epsilon})$
14: **end for**
15: Let $T_i \leftarrow T'_{i-1}$
16: **end for**
17: Return $\mathcal{C}(T_l)$

We can show that this requires $O(1/\epsilon)$ phases to execute, and each phase takes $O(1/\epsilon^2)$ rounds to compute due to Theorem 156 and our previous argument for RAKE by sibling contraction. Thus we achieve the following result:

Theorem 158. *Consider a degree-weighted tree $T = (V, E, W)$ and a problem P. Given a connected contracting function and a sibling contracting function on T with respect to P, one can compute $P(T)$ in $O(1/\epsilon^3)$ AMPC rounds with $O(n^\epsilon)$ memory per machine and $\widetilde{O}(n)$ total memory.*

Recall the definition of α-tree-contraction (Definition 161) from §15.2.1. First, we prove Lemma 167 to bound the number of phases in α-tree-contraction.

Lemma 167. *For any $\alpha \geq 2$, the number of α-tree-contraction phases until we have a constant number of vertices is bounded by $O(\log_\alpha(n))$.*

To show Lemma 167 we will introduce a few definitions. The first definition we use is a useful way to represent the resulting tree after each COMPRESS stage. Before stating the definition, recall the *Dependency Tree* T' of a tree T from Definition 165.

Definition 168. *An α-**Big-Small Tree** T' is the* dependency tree *of a tree T with weighted vertices if it is a minor of T constructed by contracting all components of T made up of low vertices v with $deg(v) < \alpha$ (i.e., the connected components of T if we were to simply remove all vertices u with $deg(v) \geq \alpha$) into a single node.*

*We call a node v in T' with $deg(v) > \alpha$ in T a **big node**. All other nodes in T', which really represent contracted components of small vertices in T, are called **small components**.*

Note that a small component may not be small in itself, but it can be broken down into smaller vertices in T. It is not hard to see the following simple property. This simply comes from

the fact that maximal components of small degree vertices are compressed into a single small component, thus no two small components can be adjacent.

Observation 169. *No small component in an α-Big-Small tree can be the parent of another small component.*

Consider our dependency tree T' based off a tree T that has been compressed. Obviously, T' is a minor of T constructed as described for α-Big-Small Tree because the weight of a vertex equals its number of children (by the assumption of Lemma 167). Note a small component refers to the compressed components, and a big node refers to nodes that were left uncompressed.

To show Lemma 167, we start by proving that the ratio of leaves to nodes in T' is large. Since RAKE removes all of these leaves, this shows that T gets significantly smaller at each step. Showing that the graph shrinks sufficiently at each phase will ultimately give us that the algorithm terminates in a small number of phases.

Lemma 170. *Let T'_i be the tree at the end of phase i. Then the fraction of nodes that are leaves in T'_i is at least $\alpha/(\alpha+4)$ as long as $w(v)$ is equal to the number of children of v for all $v \in T'_i$ and $\alpha \geq 2$.*

Proof. For our tree T'_i, we will call the number of nodes n, the number of leaves ℓ, and the number of big nodes b. We want to show that $\ell > n\alpha/(\alpha+4)$. We induct on b. When $b = 0$, we can have one small component in our tree, but no others can be added by Observation 169. Then $n = \ell = 1$, so $\ell > n\alpha/(\alpha+4)$.

Now consider T'_i has some arbitrary b number of big nodes. Since T'_i is a tree, there must be some big node v that has no big node descendants. Since all of its children must be small components and they cannot have big node descendants transitively, then Observation 169 tells

us each child of v is a leaf. Note that since v is a big node, it must have weight $w(v) > \alpha$, which also means it must have at least α children (who are all leaves) by the assumption that $w(v)$ is equal to the number of children.

Consider trimming T_i' on the edge just above v. The size of this new graph is now $n^* = n - w(v) - 1$. It also has exactly one less big node than T_i'. Therefore, inductively, we know the number of leaves in this new graph is at least $\ell^* \geq \frac{\alpha}{\alpha+4} n^* = \frac{\alpha}{\alpha+4}(n - w(v) - 1)$. Compare this to the original tree T_i'. When we replace v in the graph, we remove up to one leaf (the parent p of v, if p was a leaf when we cut v), but we add $w(v)$ new leaves. This means the number of leaves in T_i' is:

$$\begin{aligned}
\ell =& \ell^* - 1 + w(v) \\
=& \frac{\alpha}{\alpha+4}(n - w(v) - 1) - 1 + w(v) \\
=& \frac{\alpha}{\alpha+4} n - \frac{\alpha}{\alpha+4} w(v) - \frac{\alpha}{\alpha+4} - 1 + w(v) \\
=& \frac{\alpha}{\alpha+4} n + \frac{4}{\alpha+4} w(v) - \frac{2\alpha+4}{\alpha+4} \\
>& \frac{\alpha}{\alpha+4} n + \frac{4}{\alpha+4} \alpha - \frac{2\alpha+4}{\alpha+4} && (1) \\
\geq& \frac{\alpha}{\alpha+4} n && (2)
\end{aligned}$$

Where in line (1) we use that $w(v) > \alpha$ and in line (2) we use that $\alpha \geq 2$. □

Now we can prove our lemma.

Proof of Lemma 167. To show this, we will prove that the number of nodes from the start of one COMPRESS to the next is reduced significantly. Consider T_i as the tree before the ith COMPRESS and T_i' as the tree just after. Let T_{i+1} be the tree just before the $i+1$st COMPRESS, and let n_i

be the number of nodes in T_i, n'_i be the number of nodes in T'_i, and n_{i+1} be the number of nodes in T_{i+1}. Since T'_i is a minor of T_i, it must have at most the same number of vertices as T_i, so $n'_i \leq n_i$. Since T_{i+1} is formed by applying RAKE to T'_i, then it must have the number of nodes in T'_i minus the number of leaves in T'_i (ℓ'_i). Therefore:

$$n_{i+1} = n'_i - \ell'_i \leq n'_i - \frac{\alpha}{\alpha+4} n'_i = \frac{4}{\alpha+4} n'_i \leq \frac{4}{\alpha+4} n_i$$

Where we apply both Lemma 170 that says $\ell'_i \geq \frac{\alpha}{\alpha+4} n'_i$ and the fact that we just showed that $n'_i \leq n_i$. This shows that from the start of one compress phase to another, the number of vertices reduces by a factor of $\frac{4}{\alpha+4}$. Therefore, to get to a constant number of vertices, we require $\log_{\frac{\alpha+4}{4}}(n) = O(\log_\alpha(n))$ phases.

□

Now we are ready to prove our main theorem.

Theorem 158. *Consider a degree-weighted tree $T = (V, E, W)$ and a problem P. Given a connected contracting function and a sibling contracting function on T with respect to P, one can compute $P(T)$ in $O(1/\epsilon^3)$ AMPC rounds with $O(n^\epsilon)$ memory per machine and $\widetilde{O}(n)$ total memory.*

Proof. We will show that our Algorithm 21 achieves this result. Lemma 167 shows that there will be only at most $O(1/\epsilon)$ phases. In each phase i, we start by running a connectivity algorithm to find maximally connected components of bounded degree, which takes $O(1/\epsilon)$ time. Let $\mathcal{K}_{i-1}$ be the set of connected components which are leaves in T'_{i-1}. Then for each component $S_{i-1,j} \in \mathcal{K}_{i-1}$, we run BoundedTreeContract (Algorithm 20) in parallel using only our connected contracting function $\mathcal{C}$. Since the total degree of vertices over all members of $\mathcal{K}_{i-1}$ is

not larger than $|T_{i-1}|$ and the amount of memory required for storing a degree-weighted trees is not larger than the total degree, the total number of machines is bounded above by $O(n^{1-\epsilon})$. By definition, the maximum degree of any $S_{i-1,j}$ is n^{ϵ}. By Theorem 156, each instance of BoundedTreeContract requires $O(1/\epsilon^2)$ rounds, $O(|S_{i-1,j}|^{\epsilon})$ local memory and $\widetilde{O}(|S_{i-1,j}|)$ total memory. As $|S_{i-1,j}| \leq |T_{i-1}|$ (we know $|T_0| = n$ and it only decreases over time), we only require at most $O(n^{\epsilon})$ memory per machine. Since Since the total degree of vertices over all members of $\mathcal{K}_{i-1}$ is not larger than $|T_{i-1}|$, the total memory required is only $\widetilde{O}(|T_{i-1}|)) = \widetilde{O}(n)$. This is within the desired total memory constraints.

Finally, $\mathcal{R}$ is given to us as a sibling contractor. Consider the RAKE stage in our algorithm. We distribute machines across maximal leaf-stars. For any leaf-star with $n^{\epsilon} \leq deg(v) \leq kn^{\epsilon}$ for some (possibly not constant) k, we will allocate k machines to that vertex. Since again the number of vertices is bounded above by n, this requires only $O(n^{1-\epsilon})$ machines. On each machine, we allocate up to n^{ϵ} leaf-children to contract into each other. We can then contract siblings into single vertices using $\mathcal{R}$. Since there are at most n children for a single vertex, it takes at most $O(1/\epsilon)$ rounds to contract all siblings into each other. Then, finally, we can use $\mathcal{C}$ to compress the single child into its parent, which takes constant time.

Therefore, we have $O(1/\epsilon)$ phases which require $O(1/\epsilon^2)$ rounds each, so the total number of rounds is at most $O(1/\epsilon^3)$. We have also showed that throughout the algorithm, we maintain $O(n^{\epsilon})$ memory per machine and $\widetilde{O}(n)$ total memory. This concludes the proof. □

15.3.3 Simulating 2-tree-contraction in $O(1)$ AMPC rounds

Due to Theorem 158, we can compute any $P(T)$ on trees as long as we are provided with a connected contracting function and a sibling contracting function with respect to P. A natural question that arises is the following: for which class of problem P there exists black-box contracting functions? We argue that many problems P for which we have a 2-tree-contraction algorithm can also be computed in $O(1/\epsilon^3)$ AMPC rounds using n^ϵ-tree-contraction.

In many problems which are efficiently implementable in the Miller and Reif [Miller and Reif, 1989] *Tree Contraction* framework, we are given $\mathcal{C}$ and $\mathcal{R}$ contracting functions, for COMPRESS and RAKE stages respectively, which contract only one node: either a leaf in case of RAKE or a vertex with only one child in case of COMPRESS. Let us call this kind of contracting functions *unary contracting functions* and denote them by $\mathcal{C}^1$ and $\mathcal{R}^1$. This is a key point of original variants of Tree Contraction which contract odd-indexed vertices, or contract a maximal independent set of randomly selected vertices. Working efficiently regardless of using only *unary* contracting functions is the reason Tree Contraction was considered a fundamental framework for designing parallel algorithms on trees in more restricted models such as PRAM. For example, in the EREW variant of PRAM, an $O(\log(n))$ rounds tree contraction requires to use only *unary* contracting functions $\mathcal{R}^1$ and $\mathcal{C}^1$. More generally, we define i-ary contracting functions as follows.

Definition 171. *An* "i-ary contracting function"*, denoted by $\mathcal{C}^i$ or $\mathcal{R}^i$, is a* contracting function *which admits a subset $S = \{v_1, v_2, \ldots, v_k\}$ of at most $i+1$ vertices at a time such that $\sum_{j=1}^{k} deg(v_j) = O(i)$. A special case of i-ary contracting functions, are* "unary contracting functions"*, denoted by $\mathcal{C}^1$ or $\mathcal{R}^1$, which contract only one vertex at a time.*

However, in the the AMPC model, we can contract the chains more efficiently, and thus we are allowed to utilize more relaxed variants of COMPRESS stage. Furthermore, as we show in Theorem 172, designing *unary* contracting functions $\mathcal{C}^1$ and $\mathcal{R}^1$ is not easier than designing *i-ary* contracting functions $\mathcal{C}^i$ and $\mathcal{R}^i$ in the AMPC model. We show this by reducing $\mathcal{C}^i$ and $\mathcal{R}^i$ to $\mathcal{C}^1$ and $\mathcal{R}^1$ in $O(1)$ rounds for any $i = O(n^\epsilon)$. In other words, the restrictions of PRAM model, which requires $\mathcal{C}^1$ and $\mathcal{R}^1$ exclusively, enables us to directly translate a vast literature of problems solved using tree contraction to efficient AMPC algorithms for the same problem given $\mathcal{C}^1$ and $\mathcal{R}^1$.

As we have shown in Theorem 158, it is possible to solve any problem $P(T)$ in $O(1/\epsilon^3)$ AMPC rounds given a connected contracting function $\mathcal{C}$ and a sibling contracting function $\mathcal{R}$, where both are n^ϵ-ary contracting with respect to P. In what follows, we demonstrate the construction of n^ϵ-ary contracting functions given a unary connected contracting function $\mathcal{C}^1$ and a sibling contracting function $\mathcal{R}^1$.

Theorem 172. *Given a* unary connected contracting function $\mathcal{C}^1$ *and a* unary sibling contracting function $\mathcal{R}^1$ *with respect to a problem P defined on trees, one can build an* i-ary connected contracting function $\mathcal{C}^i$ *and an* i-ary sibling contracting function $\mathcal{R}^i$ *with respect to P and both $\mathcal{C}^i$ and $\mathcal{R}^i$ run in one AMPC rounds as long as $i = O(n^\epsilon)$.*

Proof. First, we present an algorithm for $\mathcal{C}^i$. We are given a connected subtree induced by $S = \{v_1, v_2, \ldots, v_k\}$ of T so that $\sum_{j=1}^{k} deg_T(v_j) = O(i)$. Since $i = O(n^\epsilon)$, the whole subtree fits into the memory of a single machine. Some of the leaves of this subtree are *known*, meaning that they are a leaf also in T, and others are *unknown*, meaning that they have children outside S. Let $\mathcal{U} = \{u_1, u_2, \ldots, u_l\}$ be the set of the children of *unknown* leaves as well as the children of

non-leaf nodes which are outside of S. Ultimately, we want to compress the data already stored on the vertices of S into a memory of $\widetilde{O}(l)$ as the degree of v_1 in the contracted tree T' will be $l+1$, and thus $deg_{T'}(v_1) = l+1$.

The $W(v_1)$ of each contracted vertex v_1 is a weighted-degreee tree structure $T^{\mathcal{C}}(v_1)$ whose leaves are the children of v_1 in T', and there is no vertex with exactly one child in $T^{\mathcal{C}}(v_1)$. Thus, the number of vertices in $T^{\mathcal{C}}(v_1)$ is bounded by $O(l) = O(deg_{T'}(v_1))$. In addition, we are guaranteed that the total size of vectors on each vertex of $T^{\mathcal{C}}(v_1)$ is bounded by $|T^{\mathcal{C}}(v_1)|$ since $\mathcal{C}^1$ and $\mathcal{R}^1$ are unary contracting functions. Therefore, we assume each $W(v_j)$ for each vertex $v_j \in S$ has stored a tree structure of size $\widetilde{O}(deg_T(v_j))$. We concatenate all these trees to get an initial $T^{\mathcal{C}}(v_1)$ whose size is bounded by $\sum_{j=1}^{k} \widetilde{O}(deg_T(v_j)) = \widetilde{O}(n^{\epsilon})$.

We run a 2-tree-contraction-*like* algorithm locally on $T^{\mathcal{C}}(v_1)$ using $\mathcal{C}^1$ and $\mathcal{R}^1$. Note that we can only rake the known leaves since the data of unknown leaves depend on their children. We repeating COMPRESS and RAKE stages until there is no known leaf or a vertex with one child remain in $T^{\mathcal{C}}(v_1)$. Then, according to Lemma 167 for $\alpha = 2$, the number of remaining vertices in $T^{\mathcal{C}}(v_1)$ is bounded by $O(l)$. We store the final $T^{\mathcal{C}}(v_1)$ in T' which requires a memory of $\widetilde{O}(l) = \widetilde{O}(w_{T'}(v_1))$. Hence, $\mathcal{C}^i$ satisfies the size-constraint on the weight vectors of the resulting weighted-degree tree.

Finally, we present an algorithm for $\mathcal{R}^i$ which is more straight-forward compared to that of $\mathcal{C}^i$. We are given a leaf-star $S = \{v_1, v_2, \ldots, v_k\}$ of T so that $\sum_{j=1}^{k} deg_T(v_j) = O(i)$. This implies that there are at most $O(n^{\epsilon})$ vertices in S as long as $i = O(n^{\epsilon})$, and we can fit the whole S into a memory of a single machine. To simulate $\mathcal{R}^i$, we only need to $k-1$ times apply $\mathcal{R}^1$ on $S_j = \{v_1, v_{j+1}\}$ at the j-th iteration. Note that every $v_j \in S$ is a leaf in T, so the data stored in $W(v_j)$ is just $\widetilde{O}(1)$ bits and not a tree structure. Theorem 172 statement is implied. □

15.3.4 Reconstructing the Tree for Linear-sized Output Problems

Consider a problem $P(T)$ whose output size is also linear in the size of input n. For instance, in maximum weighted matching we need to find the matching itself. Up to this point, in all of our algorithms, we assume the output of function $P(T)$ is of constant-sized. We simply contract the tree through some iterations until it collapses into a single vertex, and we do not need to remember anything about a vertex which is contracted as a member of a connected component or as a member of a leaf-star.

In this section, we present a general approach for retrieving the linear-sized solution in a natural scenario, where we need to retrieve a recursively-defined weight vector $P(v)$ of constant size for each vertex $v \in V$. In the special case of maximum weighted matching which can be formulated as a dynamic programming problem, $P(v)$ contains the final value of different DP values with respect to the subtree rooted at v[6].

Roughly speaking, our reconstruction algorithm is based on storing the information about components we contracted throughout the algorithm in an auxiliary memory of size $O(n)$. It is easy to observe that if we store the degree-weighted subtree of every connected component or leaf-star that we contract during the algorithm we need at most $O(n)$ addition total memory. Note that during each application of black-box contracting functions, we remove at least one vertex from the tree and each vertex except root is removed exactly once when the algorithm terminates. Namely, for every phase i we need to store $\mathcal{S}_i$ and $\mathcal{L}_i$ in Algorithm 20, and $\mathcal{K}_i$ and every $L_{i,j,z}$ in Algorithm 21 (In addition to the data stored by each black-box application of Algorithm 20). Since we have adaptive access to these subsets in AMPC, it is sufficient to index them by the id

[6]Note that retrieving $P(v)$ for each vertex v still does not give us the optimum matching and a problem-specific post-processing step is required to retrieve the actual matching

of the surviving vertex of each subset.

The full reconstruction algorithm starts after the main contraction algorithm finishes. We only need to store the information about contracted subsets during the running time of the contraction algorithm. Next, we iterate over the phases of the algorithm in reverse order, i.e., $i = \{1/\epsilon, 1/\epsilon - 1, \ldots, 1\}$, and undo the contractions that were performed during phase i. Let C_v be a connected contracted component rooted at v, and $\{w_1, w_2, \ldots, w_k\}$ be children of v post-contraction.

Whenever we undo a connected contraction like C_v, we replace v with the whole structure of C_v including $W(u)$ for every $u \in C_v, u \neq v$. Then we populate the $P(u)$ for every $u \in C_v, u \neq v$. During the contraction algorithm $P(w_j)$ is not known for any j. However, during the reconstruction we know $P(w_j)$ for every $1 \leq j \leq k$ since these vertices are contracted in a later phase than the phase we contract C_v. Hence, we have already populated $P(w_j)$ and we can use these values to locally populate $P(u)$ for every $u \in C_v$. Undoing the sibling contracting functions in much simpler since their values do not depend on other vertices nor the value of other vertices depend on their value. We populate $P(u)$ for every $u \in L$, where L is a leaf-star, based on the already constant-sized weight vectors $W(u)$.

15.4 Conclusion

This paper introduces some of the first generalized techniques for solving various problems in the AMPC model. Specifically, we show that Miller and Reif's [Miller and Reif, 1985] $O(\log n)$-time PRAM tree contraction algorithm can be efficiently extended to a constant-round low-memory AMPC algorithm. This implies $O(1/\epsilon^2)$-round algorithms for expression evalu-

ation, tree isomorphism testing, maximal matching on trees, and maximal independent set on trees. It additionally implies $O(1/\epsilon^3)$-round algorithms for maximum weighted matching and maximum weighted independent set on trees. However, we expect these algorithms to have much broader applications to tree-based problems, as did the original work by Miller and Reif.

It remains to be seen precisely which of extensions of the PRAM algorithm apply to the AMPC model. Many of them require computational overhead beyond the black-box application of the tree contraction process (which our algorithm can directly and efficiently simulate), therefore the extension of those applications to this work is highly non-trivial. Of notable interest is the application of tree contractions to graphs of bounded treewidth, where Bodlaender and Hagerup [Bodlaender and Hagerup, 1995] showed how to construct low-width tree decompositions using PRAM tree contractions. If AMPC tree contractions can also solve this and additionally solve tree contractions on graphs of bounded tree-width, then our work can be notably generalized. Another potential course of research is the exploration of problems such as MedianParent, which *cannot* be simplified to problems on trees with bounded tree width. It is an open question if these, too can be solved in $O(1/\epsilon^2)$ rounds.

Chapter 16: Adaptive Massively Parallel Algorithms for Cut Problems

16.1 Introduction

Massively Parallel Computation (MPC) – introduced by Karloff et al. [Karloff et al., 2010] in 2010 – is an abstract model that captures the capabilities of the modern parallel/distributed frameworks widely used in practice such as MapReduce [Dean and Ghemawat, 2008], Hadoop [White, 2009], Flume [Chambers et al., 2010], and Spark [Zaharia et al., 2016]. MPC has been at the forefront of the research on parallel algorithms in the past decade, and it is now known as the de facto standard computation model for the analysis of parallel algorithms.

In this paper, we focus on sublogarithmic-round algorithms for the Min Cut problem in the *Adaptive Massively Parallel Computation* ($AMPC$) model, which is a recent extension of MPC. In both MPC and $AMPC$, the input data is far larger than the memory of a single machine, and thus an input of size N is initially distributed across a collection of $\mathcal{P}$ machines. In the MPC model, the algorithm executes in several synchronous rounds, in which each machine executes local computations isolated from other machines, and the machines can only communicate at the end of a round. The total size of incoming/outgoing messages for each machine is limited by local memory constraints. We are interested in *fully-scalable* algorithms in which every machine is allocated a local memory of size $O(\mathsf{N}^{\epsilon})$ for any constant $0 < \epsilon < 1$. Moreover, we can often

improve the round complexity[1] of the massively parallel algorithms by allowing a super-linear total memory $O(\mathsf{N}^{1+\epsilon})$, for example, the filtering technique of Lattenzi et al. [Lattanzi et al., 2011] in MPC or the maximal matching algorithm of Behnezhad et al. [Behnezhad et al., 2020] in $AMPC$. So we are primarily interested in algorithms with $\widetilde{O}(\mathsf{N})$ total memory, and therefore we assume there are $\mathcal{P} = \widetilde{O}(\mathsf{N}^{1-\epsilon})$ machines. [2]

Recent developments in the hardware infrastructure and new technologies such as RDMA [Dragojevic et al., 2017], eRPC, and Farm [Dragojević et al., 2014] allow for high-throughput, low-latency communication among machines in data centers, such that remote volatile memory accesses are becoming faster than accessing local persistent storage. The concept of a shared remote memory is in particular useful when machines need to query data adaptively – i.e., deciding what to query next based on the previously queried data – which requires a communication round per query in the MPC model. Behnezhad et al. [Behnezhad et al., 2019c] incorporates this RDMA-like paradigm of remote memory access into the MPC model and introduces AMPC. In the new model, the machines can *adaptively* query from a *distributed hash table*, or a shared read-only memory, during each round. Machines are only allowed to write to shared memory at the end of each round. There is also empirical evidence that AMPC algorithms for several problems – including maximal independent set, maximal matching, and connectivity – obtain significant speedups in running time compared to state-of-the-art MPC algorithms [Behnezhad et al., 2020]. This fact, which stems from the meaningful drop in the number of communication rounds, verifies the practical power of the AMPC model.

In this paper, we provide the first AMPC-specific algorithms for the Min Cut problem. The

[1] The number of rounds is a main complexity of interest since in practice the bottleneck is often the communication phase.

[2] Where $\widetilde{O}$ hides polylogarithmic factors, i.e., $\widetilde{O}(f(n)) = O(f(n)poly\log(n))$.

Min Cut of a given graph $G = (V, E)$ is the minimum number of outgoing edges, $\delta(S)$, among every subset of vertices $S \subseteq V$. The celebrated result of Karger and Stein [Karger and Stein, 1996] solves Min Cut by recursively contracting edges in random order. Specifically, it runs two instances of the contraction process with different seeds in parallel. Each instance is run in parallel until the graph size is reduced by a factor of $\frac{1}{\sqrt{2}}$, at which point each instance recurses (thereby creating a parallel split again). They return the minimum of the two returned cuts. The algorithm itself is mainly inspired by another result of Karger [Karger, 1993] for finding the Min Cut using graph contractions. We also extend our approach to the Min k-Cut problem, in which we are given a graph $G = (V, E)$ and an integer k and we want to find a decomposition of V into k subsets $V_1, V_2, \ldots, V_k$ so that $\sum_{i=1}^{k} \delta(V_i)$ is minimized. We utilize the greedy algorithm of Saran and Vazirani [Saran and Vazirani, 1995] which gives an $O(2 - \frac{2}{k})$-approximation of the Min k-Cut. Gomuri and Hu give an alternative algorithm with the same approximation guarantee with additional features [Gomory and Hu, 1961].

We study the Min Cut and Min k-Cut problems in the $AMPC$ model. We give $O(\log \log n)$-round $AMPC$ algorithms for a $(2 + \epsilon)$-approximation of Min Cut and a $(4 + \epsilon)$-approximation of Min k-Cut.

16.1.1 Adaptive Massively Parallel Computation ($AMPC$)

Massively Parallel Computation (MPC) and Adaptive Massively Parallel Computation ($AMPC$) both sprung out of an interest in formalizing a theoretical model for the famous MapReduce programming framework. The most common problems in MPC and AMPC are on graph inputs, and since our paper only considers graph problems, we define these two models

in terms of problems on graphs. Consider a graph $G = (V, E)$ with $n = |V|$ and $m = |E|$.

In standard MPC [Goodrich et al., 2011, Andoni et al., 2014, Karloff et al., 2010, Lattanzi et al., 2011], we are given a collection of $\mathcal{P}$ machines and are allowed to compute the solution to a problem in parallel. As we have already discussed, MPC computation occurs in synchronous rounds, each consisting of local polynomial-time computation and ending with machine-machine communication where all messages sent to and from a machine must fit within its local memory. Fully-scalable algorithms, the strongest memory regime in MPC, require the local memory to be constrained by $O(n^\epsilon)$ for any given $0 < \epsilon < 1$. Additionally, we are primarily interested in algorithms that require at most $O(\log n)$ rounds. However, often sublogarithmic – i.e., $O(\sqrt{\log n})$ or $O(\log \log n)$ – round complexity is much more desirable. In most cases, the total space must be at most $\widetilde{O}(n + m)$, though sometimes we allow slightly superlinear total space.

AMPC extends MPC to add functionality while remaining implementable on modern hardware. Formally, in the AMPC model, we are given a set of distributed hash tables $\mathcal{H}_0, \ldots, \mathcal{H}_k$ for each of the k rounds of computation. These hash tables are each limited in size by the total space of the model (i.e., $\widetilde{O}(n + m)$). As in MPC, we are given a number of machines and computation proceeds in rounds. In each round, local computations occur and then messages are sent between machines. The distinction in AMPC is that during the local computations, machines are allowed simultaneous read access to the hash table for that round (i.e., $\mathcal{H}_{i-1}$ for round i) and during the messaging phase of the round, they are allowed to write data to the next hash table, $\mathcal{H}_i$. Reading and writing is limited by machine local memory. The power of the AMPC model over the MPC model is that, at the beginning of a round, the machines do not need to choose all the data they will access during the round. Instead, they can dynamically access the data stored in the hash table over the course of the local computation, thus potentially selecting data based on its own

local computation.

It is not too hard to see that AMPC is a strictly stronger model than MPC. In fact, it was formally shown that all MPC algorithms can be implemented in AMPC with the same round and space complexities [Behnezhad et al., 2019c].

16.1.2 Our Contributions and Methods

This work is the first to study the Adaptive Massively Parallel Computation (AMPC) model for Min Cut problems on graphs. We mainly focus on the standard single Min Cut problem, although we also propose an approximation algorithm for the Min k-Cut problem. Our main result for the Min Cut problem is a $2+\epsilon$ approximate algorithm that uses sublogarithmic $O(\log\log n)$ rounds.

Theorem 173. *There is an $O(\log\log n)$-round AMPC algorithm that uses $\widetilde{O}(n+m)$ total memory and $\widetilde{O}(n^{\epsilon})$ memory per machine which finds a $(2+\epsilon)$-approximation of Min Cut with high probability.*

Note that this is a vast improvement over the current state-of-the art algorithms in MPC by Ghaffari and Nowicki [Ghaffari and Nowicki, 2020], which achieves the same $2+\epsilon$ approximation in $O(\log n \log\log n)$ rounds. Both our algorithm and that of Ghaffari and Nowicki use Karger's methods as a general structure for finding the Min Cut. Using this method, the goal is to recursively execute random graph contractions. From the results of Karger, the contraction process either finds a singleton cut that is a $2+\epsilon$ approximation or preserves a specific Min Cut with probability dependent on the depth of recursion. To leverage this result, at each step of the recursion process, we find the best singleton cut on the existing graph. Once the graph is small

enough, the problem can be solved efficiently. Out of all the singleton solutions found during this process and the final Min Cut on the small graph, we simply select the best cut. This is a $2+\epsilon$ approximate Min Cut with high probability.

To implement this approach in a distributed model, both methods assign random weights to the edges of the input graph and find a minimum spanning tree (MST). Greedily, selecting edges in order of decreasing weight, we contract the graph along the current edge. This process is equivalent to the same greedy random contraction process on the original graph. This step, already, currently requires at least $\Omega(\log n)$ rounds in MPC, but the flexibility of the AMPC model allows us to achieve this step in a constant number of rounds.

It remains to show how can one find the best singleton cuts at each level of recursion. In order to do this, we employ a *low-depth tree decomposition* on the minimum spanning tree until it becomes a set of separated vertices. On top of this recursive divide-and-conquer process, we design a process to compute and remember the best singleton cut.

The high level idea of recursively partitioning the tree and applying a process on top of that to find the best singleton cut is the same in both our paper and Ghaffari and Nowicki's paper [Ghaffari and Nowicki, 2020]. However, the processes used to do this in MPC do not yield simple improvements in AMPC. Rather, we must use entirely novel techniques that leverage adaptivity to get truly sublogarithmic results. In fact, this must be done in constant rounds to achieve our results, whereas Ghaffari and Nowicki do this in $O(\log n)$ rounds. In order to create a tree decomposition, we consider maximal paths of heavy edges (i.e., edges that go from a parent to its child with the largest subtree). These paths are replaced by binary trees whose leaves are the path and the root connects to the path's parent. Consider labeling the resulting vertices in the graph with their depth. For each internal node in one of these binary trees, which was *not* a

vertex in the original tree, we select a specific descendant leaf in the binary tree expansion of the path to send its depth to. The final value a vertex receives is then what we call the "label", which measures at what level of recursion the tree splits at that vertex. An entire labeling of the tree encodes an entire tree decomposition. This is done in constant AMPC rounds.

To compute the singleton cuts at each level, we assign to each singleton cut formed during the contraction process a vertex that has the lowest label. We show that such vertices are well-defined, i.e. there is only one vertex with the lowest label within vertices on the same side of a singleton cut. Because removing vertices of labels lower than i partitions the tree into disjoint subtrees such that each subtree contains at most one vertex with label i, we are able to calculate minimal singleton cuts corresponding to these vertices with label i in parallel in a constant number of AMPC rounds. Since, we constructed the low-depth decomposition such that the range of labels has size $O(\log^2 n)$, thus, by increasing the total memory, we can perform these computations for all different lables in a constant number of AMPC rounds. For more details, we defer to Section 16.4.

We then show how this work can be leveraged to achieve efficient results for approximate Min k-Cut, generalizing the results from Saran and Vazirani [Saran and Vazirani, 1995]. At a high level, we start by computing a Min Cut. Then we add the edges of the cut to a set D. In all following $k-1$ iterations, we calculate the Min Cut on the graph without edges in D, and add the new cut edges to D for the next iteration. The set of the first k cuts we compute is our k-cut.

Compared to Saran's and Vazirani's technique, our method uses an *approximate* Min Cut rather than an exact Min Cut on each splitting step. This requires adapted analysis of this general approach. We employ the structure of Gomory-Hu trees (see [Gomory and Hu, 1961]) for this purpose and show the following result:

Theorem 174. *Algorithm APX-SPLIT is an* $(4+\epsilon)$ *approximation of the Min* k*-Cut. Furthermore, it can be implemented in the AMPC model with* $\mathrm{O}(n^{\epsilon})$ *memory per machine in* $\mathrm{O}(k \log\log n)$ *rounds and* $\mathrm{O}(m)$ *total memory.*

Therefore, for small values of k, we can achieve efficent algorithms for $4+\epsilon$ approximate Min k-Cut in AMPC. Note that there are no existing results in the MPC model, however our methods applied to the work of Ghaffari and Nowicki [Ghaffari and Nowicki, 2020] yield:

Corollary 175. *There is an algorithm that achieves a* $(4+\epsilon)$ *approximation of the Min* k*-Cut with high probability that can be implemented in the MPC model with* $\mathrm{O}(n^{\epsilon})$ *memory per machine in* $\mathrm{O}(k \log n \log\log n)$ *rounds and* $\mathrm{O}(m)$ *total memory.*

Note there is still a logarithmic-in-n improvement in the round complexity in AMPC over MPC no matter the value of k. Due to space constraints both these result are presented in the appendix.

16.2 Minimum Cut in AMPC

Karger and Stein [Karger and Stein, 1996] proposed a foundational edge contraction strategy for solving Min Cut:

- Create two copies of G, and independently on each, contract edges in a random order until there are at most $\frac{n}{\sqrt{2}}$ vertices.
- Recursively solve the problem on each contracted copy until they have constant size.
- Return the minimum of the cuts found on both copies.

Lemma 176 ([Karger and Stein, 1996])**.** *The contraction process executed to the point where there are only $\frac{n}{t}$ vertices left preserves any fixed minimum cut with probability $\Omega\left(\frac{1}{t^2}\right)$.*

According to Lemma 176, naively contracting random edges until there are only two vertices remaining preserves at least one minimum cut with probability $\Omega\left(\frac{1}{n^2}\right)$. Thus, we need to repeat the naive contraction process at least $O\left(n^2 \log n\right)$ times so that we have a high *probability of success*, i.e., preserving a minimum cut. However, Karger and Stein [Karger and Stein, 1996] show that their recursive strategy succeeds with probability $\Omega\left(\frac{1}{\log n}\right)$. In turn, running $O\left(\log^2 n\right)$ instances of the recursive strategy is enough to find a minimum cut with high probability.

Roughly speaking, the choice of $t = \sqrt{2}$ as the inverse of the branching factor assures that a minimum cut is preserved with probability $\frac{1}{t^2} = \frac{1}{2}$ throughout the contractions in each copy. Thus, the probability of success, say $P(n)$, for n vertices is bounded by:

$$P(n) \geq 1 - \left(1 - \frac{1}{2} \cdot P\left(\frac{n}{\sqrt{2}}\right)\right)^2 \tag{16.1}$$

Note that the random contractions in two copies are assumed to be independent, and the probability of success for each copy is at least $\frac{1}{2} \cdot P\left(\frac{n}{\sqrt{2}}\right)$ since we recurse on the resulting contracted graph with $\frac{n}{\sqrt{2}}$ vertices. Inequality (16.1) implies that at the k-th level of recursion (counting from the bottom), the probability of success is $\Omega\left(\frac{1}{k}\right)$, and in particular $\Omega\left(\frac{1}{\log n}\right)$ at the root of recursion. [Karger and Stein, 1996].

Let us now give some high-level insight into the approach by Ghaffari's and Nowicki. Ghaffari and Nowicki [Ghaffari and Nowicki, 2020] observed that if we only desire a $(2+\epsilon)$ approximate cut, we can use a better bound for the probability of preserving a minimum cut, or alternatively, the success probability.

Lemma 177 ([Ghaffari and Nowicki, 2020, Karger and Stein, 1996]). *On an n-vertex graph G, let C be a minimum cut with weight λ. Fix an arbitrary $\epsilon \in (0,1)$. The described random contraction process that contracts G down to $\frac{n}{t}$ vertices either at some step creates a singleton cut of size at most $(2+\epsilon)\lambda$ or preserves C - i.e., it does not contract any of its edges - with probability at least $\frac{1}{t^{1-\epsilon/3}}$.*

A *singleton cut* is a partitioning of graph vertices so that there is only one vertex on one side, i.e., $\delta(S)$ so that $|S| = 1$. Assuming that one is able to verify whether a singleton cut of a small size has been formed during the contraction process, they show that this greater probability of success can boost the recursive process. In short, consider the k-th level of recursion, where level 0 corresponds to the bottom level. Let $\frac{n}{t_k}$ be the size of a single recursive instance at level k, and denote by s_k the total number of instances on this level. For all k, they ensure $s_k = t_k^{1-\epsilon/3}$.

Now, let $x_k^{1-\epsilon/3}$ be the branching factor on level k. That is, the recursion produces $x_k^{1-\epsilon/3}$ copies of the instance at level k, and on each of them independently contracts edges in a random order until the number of vertices is bigger than $\frac{n}{t_k} \cdot \frac{1}{x_k}$. If we have an algorithm that is able to *track* whether a small singleton cut appeared in each of these random processes, we either get a singleton cut that $(2+\epsilon)$ approximates a minimum cut or a minimum cut is preserved with probability $x_k^{1-\epsilon/3}$. Since we made $x_k^{1-\epsilon/3}$ copies, by a similar argument as in Karger's approach, we get that, in the latter case, the probability of preserving a minimum cut is $\Omega\left(\frac{1}{k}\right)$.

Finally, observe that on the k-th level of recursion, the most costly operation is copying a k-th level instance $x_k^{1-\epsilon/3}$ times in order to contract edges in each of these instances. Since the instance has size $\frac{n}{t_k}$ and we have s_k instances, processing these tasks in parallel requires $\frac{n}{t_k} \cdot s_k \cdot x_k^{1-\epsilon/3}$ space. If one want to fit this in $O(n)$ space, then it must be that $x_k \leq t_k^{(\epsilon/3)/(1-\epsilon/3)}$.

Anyway, we get that the number of contractions we can make on k-th level is polynomial in the number of contractions we made on higher levels, and if the recurrence is solved, then it follows that it will be $O(\log \log n)$ levels until we reach a graph of a constant size.

Ghaffari and Nowicki [Ghaffari and Nowicki, 2020], use Lemma 177 and the above boosting scheme to show an $O(\log \log n \cdot \log n)$-round MPC algorithm for Min Cut. The main non-trivial part of their algorithm involves tracking the smallest singleton cut on each recursion level, which they do in $O(\log n)$ rounds because of the divide and conquer nature of their approach. Effectively, they assign all edges random and unique edge weights, and contract all uncontracted edges in decreasing order by edge weight. It can then be shown that all that needs to be done is to compute the MST of this graph and contract these edges accordingly (all other edges will be automatically contracted when another edge is contracted). We reduce the number of rounds for singleton cut tracking down to $O(1)$ rounds in the AMPC model. We aim to prove the following theorem.

Theorem 173. *There is an $O(\log \log n)$-round AMPC algorithm that uses $\widetilde{O}(n+m)$ total memory and $\widetilde{O}(n^{\epsilon})$ memory per machine which finds a $(2+\epsilon)$-approximation of Min Cut with high probability.*

To track singleton cuts, the first step is to find a *low depth decomposition* of the current MST. At a high level, a low depth decomposition of a tree is a labeling of its vertices with values 1 through d, where d is the depth. This label must satisfy the following: for every level $i \in [d]$, the connected components induced on vertices with label at least i must contain at most one vertex for each i. This defines a recursive splitting process: starting at depth 1, there must be at most one vertex v with the minimum label, so we can split the tree into multiple parts by removing v. Then

we simply recurse on each connected component, considering the next set of labels, and knowing the process will always split each connected component once at a time. This is the general idea captured by both this and previous works. However, in order to increase the efficiency of this step, we require a new decomposition structure (see Definition 178) and new methods for finding the decomposition. Notice that it is always true that at each level, each connected component contains at most one vertex at the next level.

In Section 16.3, we show how to find a low depth decomposition with depth $O(\log^2 n)$ in AMPC in $O(1/\epsilon)$ rounds (Lemma 179) with $O(n^\epsilon)$ space per machine. Roughly speaking, we create a heavy-light decomposition of the MST, where we store "heavy paths" consisting of edges connecting vertices to their children with the largest number of descendants and isolated "light nodes". We replace each heavy path with a complete binary tree whose leaves contain the vertices in the path, which gives us an efficient structure to obtain our labeling. This yields our low depth decomposition.

In the next step, we compute the size of of every singleton cut S that is created during the process. Note that the contractions are inherently sequential and the number of contractions we need to make at step k is $x_k \in O(n)$. However, each singleton cut is a connected component on the MST containing a specific edge e, whose contraction – in the increasing order of contracting MST edges – results in subset S, if we only allow the edges that have a smaller weight than e. We partition these connected components based on the vertex in the cut with the lowest level in the heavy-light decomposition of the MST. This way, we can compute every level of the low depth decomposition in parallel with only an $O(\log^2 n)$ blowup in total memory. In Section 16.4, we show that we can track every singleton cut in the contraction process in $O(1)$ AMPC rounds. A high level pseudocode of the main algorithm is given in Algorithm 22.

Algorithm 22 AMPC-MinCut
(An algorithm that calculates $(2+\epsilon)$ approximation of Min Cut in G. The novel part is underlined.
)

Input A graph $G = (V(G), E(G))$, a parameter k
Output $(2 + \epsilon)$ approximation of Min Cut.

1: **if** $|G| \in n^{\epsilon}$ **then**
2: Return Min Cut of G calculated on a single machine
3: **end if**
4: Let $\widehat{G_1}$, ..., $\widehat{G_k}$ be copies of G with assigned random weight on edges (independently for each copy)
5: In parallel for all $i \in [k]$, $S_i \leftarrow$ MinSingletonCut$(\widehat{G_i})$
6: In parallel for all $i \in [k]$, $G_i \leftarrow$ copy of $\widehat{G_i}$ after first k contractions
7: In parallel, $C_i \leftarrow$ AMPC-MinCut(G_i)
8: $\min(S_1, \ldots, S_k, C_1, \ldots, C_k)$

Note that MinSingletonCut (Algorithm 24) is introduced in Section 16.4 and it leverages LowDepthDecomp (Algorithm 23) from Section 16.3.

16.3 Generalized Low Depth Tree Decomposition

This section and the next address our algorithmic formulation and analysis. Note that all omitted proofs are deferred to the Appendix.

In order to efficiently compute the singleton cuts in parallel, we first need to compute an efficient decomposition of the MST. The low depth tree decomposition Ghaffari and Nowicki [Ghaffari and Nowicki, 2020] introduce is a very specific decomposition with i levels such that at each level ℓ, any connected component of size s on vertices at that level or higher has a single vertex at level ℓ that separates the component into two components with size at least $s/3$ each. Unfortunately, it is unclear how to calculate this precise decomposition efficiently in AMPC. To work around this, we introduce a more generalized version of the low depth tree decomposition, show that it can be computed in AMPC, and later show that we can leverage this to obtain our

Min Cut algorithm.

Definition 178. *A **generalized low depth tree decomposition** of some tree T is a labeling $\ell : V(T) \to [h]$ of vertices with **levels** for decomposition height $h \in O(\log^2 n)$ such that for each level i, the connected components induced on $T^i = \{v \in T : \ell(v) \geq i\}$ have at most one vertex labeled i each.*

Notice we do not define how a level is assigned; we simply require it is assigned to satisfy the property on connected components. We describe one way to do that in this section.

To see what such a decomposition looks like, consider a process where at timestep t we look at the subgraph induced on the vertices v with $\ell(v) \geq t$ (i.e., T^t). Consider a connected component C and let v be its minimum level vertex. Then $\ell(v) \geq t$, and it is the only vertex at that level in C. At timestep $\ell(v) + 1$, C becomes separated into multiple components who all contain a vertex adjacent to v. This process defines forests with smaller and smaller trees as time passes, and eventually results in isolated vertices. The completion time of this process depends on the height of the decomposition, which in our case is $O(\log^2 n)$. We will, of course, make this more efficient in Section 16.4.

It is not that hard to see that Ghaffari and Nowicki's low depth tree decomposition is a specific example of generalized tree decomposition with depth $O(\log n)$. They put a single vertex in the first level and then simply recurse on the two trees in the remaining forest. Note that they require additional properties of this decomposition to obtain their result, specifically that each new component has size at least $\frac{1}{3}$ of the original component, but we will see later that these are not necessary for finding the singleton cuts.

Like Ghaffari and Nowicki in MPC, we prove this can be computed efficiently in, instead,

AMPC.

Lemma 179. *Computing a generalized low depth tree decomposition of an n-vertex tree can be done in $O(1/\epsilon)$ AMPC rounds with $O(n^\epsilon)$ memory per machine and $O(n \log^2 n)$ total memory.*

The rest of this section is dedicated to proving Lemma 179. The formal and complete algorithm is shown in Algorithm 23 and further details and definitions can be found later in this section. At a high level, our algorithm proceeds as follows:

1. Root the tree and orient the edges [line 2].

2. Contract heavy paths in a *heavy-light decomposition* of T into *meta vertices* to construct a *meta tree*, T_M [lines 3 to 5].

3. For each meta vertex, create a *binarized path*, a binary tree whose leaves are the vertices in the heavy path, in order. Expanding meta vertices in this manner yields our *expanded meta tree* [lines 7 to 11].

4. Label each vertex according to properties of the expanded meta tree. For all new vertices (i.e., vertices created in step 3) v, label v with the depth of the highest vertex u in the same meta node such that v is the leftmost leaf descending from the rightmost child of u in the binarized path of the meta node [lines 13 to 15].

Each of these steps correspond to the following subsections. For instance, step 1 corresponds to Section 16.3.1. All relevant terminology related to these steps are additionally found in the corresponding subsections. Lemma 179 is proven at the end of the final subsection.

Algorithm 23 LowDepthDecomp
(Computing a generalized low depth tree decomposition of an input tree in AMPC)

Input A tree $T = (V(T), E(T))$.
Output A mapping $\ell : V(T) \to \mathbb{N}$ of tree vertices to levels.

1: Initialize $\ell : V(T) \to \mathbb{N}$
2: Root and orient T
3: Let $T_H = (V(T), \{e \in E(T) : e \text{ is heavy}\}$
4: Let $\mathcal{P}$ be the connected components of T_H
5: Let $T_M = (\mathcal{P}, \{(P_1, P_2) : P_1, P_2 \in \mathcal{P}, \exists (u_1, u_2) \in V(P_1) \times V(P_2)$ such that $(u_1, u_2) \in E(T)\})$
6: **for** $v \in T_M$ of heavy path P_v in parallel **do**
7: Let $V(T_v)$ be a vertex set of size $2|P_v| - 1$ with associated indices $1, \ldots, 2|P_v| - 1$, denoted by i_u
8: Let $T_v = (V(T_v), \{(u, p_u) : i_{p_u} = \lfloor i_u/2 \rfloor\})$
9: Pre-order traverse T and sort P_v accordingly
10: Pre-order traverse T_v and let L be its sorted leaves
11: For all $i \in [|P_v|]$, map $P_v[i]$ to $L[i]$
12: **for** $u \in V(T_v)$ **do**
13: Find path P^u to the root of the expanded meta-tree
14: Let $u' \in V(T_v) \cap P^u$ be such that u is the leftmost descendant of u''s right child (otherwise $u' = u$)
15: Label $\ell(u) = d(u')$
16: **end for**
17: **end for**
18: Return ℓ limited to the original vertices in T

16.3.1 Rooting the Tree

Like in Ghaffari and Nowicki, the first thing we need to do in line 2 of Algorithm 23 is compute an orientation of the edges. Fortunately this, along with rooting the tree, can be done quickly in AMPC by the results of Behnezhad et al. [Behnezhad et al., 2019c] in their Theorem 7.

Lemma 180 (Behnezhad et al. [Behnezhad et al., 2019c])**.** *Given a forest F on n vertices, the trees in F can be rooted and edges can be oriented in $O(1/\epsilon)$ AMPC rounds w.h.p. using $O(n^\epsilon)$ local memory and $O(n \log n)$ total space w.h.p.*

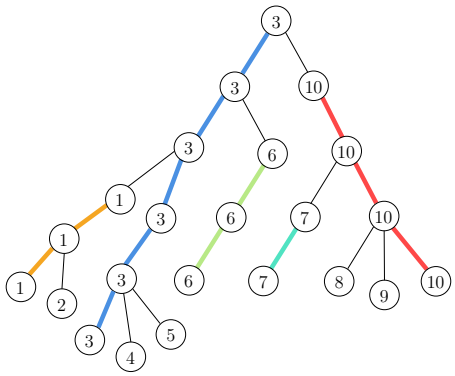


Figure 16.1: The heavy-light decomposition of an example tree.

Here, w.h.p. means "with high probability." This completes the first step of our algorithm.

16.3.2 Meta Tree Construction

We also leverage Ghaffari and Nowicki's notion of heavy-light decompositions for our AMPC algorithm, which can be found from lines 3 through 5 in Algorithm 23. This process allows us to quickly decompose the tree into a set of disjoint paths of *heavy edges*, which are defined as follows (note that our definition slightly deviates from Ghaffari and Nowicki [Ghaffari and Nowicki, 2020], where the heavy edge must extend to the child with the largest subtree without requiring this subtree to be that large, though it is the same as the definition used by Sleator and Tarjan [Sleator and Tarjan, 1981]):

Definition 181 (Sleator and Tarjan [Sleator and Tarjan, 1981])**.** *Given a tree T and a vertex $v \in T$, let $\{u_i\}_{i \in k}$ be the set of children of v where the subtree rooted at u_1 is the largest out of all u_i. If there is no strictly largest subtree, we arbitrarily choose exactly one of the children with a largest subtree. Then (u_1, v) is a **heavy edge** and (u_i, v) is a **light edge** for all $1 < i \leq k$.*

Then the definition of a heavy path follows quite simply.

Definition 182 (Ghaffari and Nowicki [Ghaffari and Nowicki, 2020])**.** *Given a tree T, a* ***heavy path*** *is a maximal length path consisting only of heavy edges in T.*

Ghaffari and Nowicki then make the observation that the number of light edges and heavy paths is highly limited in a tree. This comes from a simple counting argument, where if you consider the path from root r to some vertex v, any time you cross a light edge, the size of the current subtree is reduced by at least a factor of 2. This holds even with our different notion of heavy edges since subtrees rooted at children of light edges are still much smaller compared to the subtree rooted at the parent vertex. This bounds the number of light edges between r and v, where each pair of light edges are separated by at most one heavy path, and therefore it also bounds the number of heavy paths.

Observation 183 (Ghaffari and Nowicki [Ghaffari and Nowicki, 2020])**.** *Consider a tree T oriented towards root r. For each vertex v, there are only $O(\log n)$ light edges and only $O(\log n)$ heavy paths on the path from v to r.*

Using the definition of heavy edge from Sleator and Tarjan [Sleator and Tarjan, 1981] instead of from Ghaffari and Nowicki, we get an additional nice property. This is because in our definition, every internal vertex has one descending heavy edge to one child.

Observation 184 (Sleator and Tarjan [Sleator and Tarjan, 1981])**.** *Given a tree T and an internal vertex $v \in T$, v must be on exactly one heavy path. For a leaf $\ell \in T$, ℓ must be on at most one heavy path.*

Our first goal is to compute what we call a *meta tree*. This is a decomposition of our tree that will allow us to effectively handle heavy edges. It is quite analogous to Ghaffari and

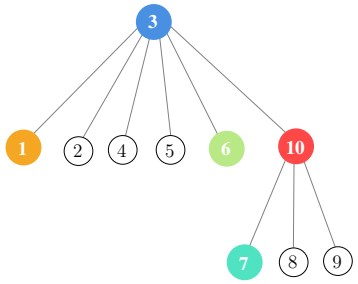


Figure 16.2: The *meta-tree* of the same tree from Figure 16.1 is demonstrated in this figure.

Nowicki's notion of the heavy-light decomposition, which partitions the tree into heavy and light edges.

Definition 185. *Given a tree T, the **meta tree** of T, denoted T_M, comes from contracting all the heavy paths in T. We call the vertices of T **original vertices** and the vertices of T_M **meta vertices**.*

Note that contracting all heavy paths simultaneously is valid because, by Observation 184, all heavy paths must be disjoint. Additionally, all internal meta vertices are contracted heavy paths (as opposed to original vertices), again by Observation 184. We note that in AMPC, since connectivity is easy, it is additionally quite easy to contract the heavy paths of T into single vertices.

Lemma 186. *Given a tree T, the meta tree T_M can be computed, rooted, and oriented in AMPC in $O(1/\epsilon)$ rounds with $O(n^\epsilon)$ memory per machine and $O(n \log^2 n)$ total space w.h.p.*

This completes the second step of our decomposition algorithm.

16.3.3 Expanding Meta Vertices

In order to label the vertices, we need a way to handle the heavy paths corresponding to each meta vertex. Let $v \in T_M$ be a meta vertex, and P_v be the heavy path of original vertices in

T corresponding to v. Note that we have no stronger bound on the length of a heavy path than $O(n)$. Therefore, a recursive partitioning, or labeling of vertices that has polylogarithmic depth must be able to cleverly divide heavy paths. We can do this with a new data structure.

Definition 187. *Given some path P, a **binarized path** is an almost complete binary tree T with $|P|$ leaves where there is a one-to-one mapping between P and the leaves of T such that the pre-order traversal of P and T limited to its leaves agree.*

By "agree", we mean that if a vertex $v \in P$ comes before a vertex $u \in P$ in the pre-order traversal of P, then it also does in the pre-order traversal of T. To characterize this tree, we make a quick observation:

Observation 188. *An almost complete binary tree on n leaves has $2n-1$ vertices, $\lfloor \log_2 n \rfloor + 1$ max depth, and every layer is full except the last, which has $2n - 2^{\lfloor \log_2 n \rfloor + 1}$ vertices.*

Additionally, we can find a relationship between the ancestry of triplets in P based off of the order of the three vertices. While this is not required for expanding meta vertices, it is a property of the binarized path that will be useful when we label vertices later.

Observation 189. *Given a binarized path T of a path P, for any $u, u', u'' \in P$ that appear in that order (or reversed), if v is the lowest common ancestor of u and u' and v' is the lowest common ancestor of u and u'', then v' is an ancestor of v or $v' = v$.*

To create the tree, we do the following for every $v \in T_M$:

1. Create an almost complete binary tree T_v with $|P_v|$ leaves, linking children to parents and noting if a vertex is a left or right child [lines 7 and 8].

2. Do a pre-order traversal of T_v and P_v and map the vertices in P_v to the leaves of T_v such that the pre-order traversal of P_v and of T_v limited to its leaves agree. [lines 9 to 11].

Next, it is pretty direct to see that the produced tree is a binarized path.

Observation 190. *The process described above produces a binarized path T_v of P_v for all v.*

We prove that this can be done in the proper constraints.

Lemma 191. *The heavy paths of a tree can be converted into binarized paths in $O(1/\epsilon)$ AMPC rounds with $O(n^\epsilon)$ local memory and $O(n \log n)$ total space w.h.p.*

16.3.4 Labeling Vertices

Our next goal is to label the vertices with the level they should be split on. Consider, hypothetically, expanding the meta tree T_M such that every heavy path for a meta vertex v is replaced with its binarized path (which is an almost complete binary tree) T_v, and the tree continues at the leaves corresponding to the nodes in the heavy path. Note that only some vertices in the hypothetical tree correspond to vertices in the original tree T. Specifically, the internal nodes of each component subtree T_v are not vertices in T, but the leaves correspond exactly to the vertices in T.

Ultimately, for a vertex $u \in T$ in meta vertex v, let u' be the vertex in T_v such that u is the leftmost leaf-descendant of the right child of u' in T_{u_M} (or if this doesn't exist, $u' = u$). Then we will label $\ell(u) = d(u')$ where d is the depth in the expanded meta tree. Following this, our vertex labeling process will be as follows for each $v \in T_M$ and $u \in T_v$:

1. Vertex u finds the path P^u from u to the root of T_M, assuming the meta vertices are expanded [line 13].

2. Let u' be the highest vertex in T_v such that u is the leftmost descendant of the right child of u. If there is no such vertex, let $u' = u$ [line 14].

3. Label u with the depth (assuming roots have depth 1) of u' in the expanded T_M [line 15].

We start by making a quick observation that comes directly from Observations 183 and 188.

Observation 192. *The max depth of T_M with meta nodes expanded ("the expanded T_M") into binary trees is $O(\log^2 n)$.*

This will be greatly helpful in showing the efficiency of our algorithm. We now show that this final part can be implemented efficiently, which is sufficient to prove our main lemma.

Lemma 193. *The process described above finds a generalized low depth tree decomposition of original tree T of height $h \in O(\log^2 n)$ in 1 round with $O(n^\epsilon)$ local memory and $O(n \log^2 n)$ total space.*

16.4 Calculating the smallest singleton cut

In this section, we show a $O(1/\epsilon)$ round AMPC algorithm that executes a series of contractions and outputs the size of the smallest singleton cut that appeared during the contraction process. That is we prove the following result.

Theorem 194. *There exists an AMPC algorithm that given a graph G with unique weights on edges calculates the minimum singleton cut that appears during the contraction process in $O(1/\epsilon)$ rounds using $O(n^\epsilon)$ local memory and $O((n+m)\log^2 n)$ total space.*

16.4.1 Contraction process

We view the contraction process of a weighted graph $G = (V, E, w : E \to [n^3])$ as a sequential process in which we iterate over multiple timesteps 0 to n^3. For a given time i, we contract the edge e having $w(e) = i$ to a single vertex. Let $G_0, \ldots, G_{n^3}$ be the sequence of graphs created in the process, where G_0 denotes the graph before any contraction and G_{n^3} denotes the graph after all contractions. Via a quick comparison to Kruskal's algorithm, it is clear that the edges whose contraction changed the topology of the graph must belong to the minimum spanning tree of the weighted graph G (since weights are unique, the MST is unique as well). Let $T = (V, E_T, w : E_T \to [n^3])$ be the minimum spanning tree of G.

From the previous observation, it is enough to consider only contracting edges from tree T, which we will focus on in the rest of this section. It will also be convenient visualize vertices as simply being grouped instead of fully contracted.

Definition 195. *A **bag** of vertex v at time $t \in [n^3]$, which we denote $\mathsf{bag}(v, t)$, is the set of vertices that can be reached from v using only edges of tree T of weight at most t. We denote $\mathsf{nbr\_bag}(v, t)$ for set of **neighbors of a bag**, that is set of these vertices u that do not belong to the bag and there exists an edge connecting u and any vertex of the bag of weight greater than t. The **degree of a bag**, denoted $\Delta\mathsf{bag}(v, t)$, is the size of the set $\mathsf{nbr\_bag}(v, t)$.*

If we proceed with our edge contraction process, where an edge with weight t is contracted at time t, then $\mathsf{bag}(v, t)$ is the set of all vertices that have been contracted with v at time t. The value $\Delta\mathsf{bag}(v, t)$ is simply the degree of the vertex that corresponds to contracted vertices. Therefore, the following simple observations holds.

Observation 196. *The value of the minimum singleton cut in the contraction process of the weighted graph G is equal to the following:*

$$\min_{v \in V, t \in [n^3]} \Delta\mathsf{bag}(v, t).$$

16.4.2 Simulating tree contractions with low depth decomposition

By Observation 196 our goal is to calculate the value of

$$\min_{v \in V, t \in [n^3]} \Delta\mathsf{bag}(v, t).$$

To find this, we could calculate the value $\min_{t \in [n^3]} \Delta\mathsf{bag}(v, t)$ for every vertex v independently in parallel. However, this would require a minimum of $\Omega(n \cdot (n + m))$ total space, which roughly corresponds to replicating the whole graphs for each independent instance. There are two key observations that will allow us to reduce the space complexity. First, bags are determined solely from the topology of tree T. Second, for larger t, it is likely the case that $\mathsf{bag}(u, t) = \mathsf{bag}(v, t)$, so we would like to remove this redundant computation. Therefore, we will exploit tree properties and the low depth decomposition to partition the work and avoid redundancy.

Let $\ell : V \to [h], h \in O(\log^2 n)$ be the labeling from the generalized low depth decomposition of tree T (see Definition 178). Let us asses to each bag a uniquely chosen vertex.

Definition 197. *The **leader** of a bag, denoted* $\mathsf{bagLeader}(v, t)$*, is the vertex u with the smallest label $\ell(u)$ among all vertices from* $\mathsf{bag}(v, t)$*. We define a number* $\mathsf{ldr\_time}(v)$ *to be the greatest number $0 \leq t' \leq n^3$ such that* $\mathsf{bagLeader}(v, t') = v$*.*

Let us first argue the correctness of the above definitions.

Lemma 198. *The leader of every bag can be determined uniquely. Also, for every vertex $v \in V$ it holds: the number* $\mathsf{ldr\_time}(v)$ *exists,* $\mathsf{ldr\_time}(v) \geq 0$*, and for every* $0 \leq t' \leq \mathsf{ldr\_time}(v)$ *we have that* $\mathsf{bagLeader}(v, t') = v$.

Using the fact that each bag has exactly one leader, we can reformulate the expression $\min_{v \in V, t \in [n^3]} \Delta\mathsf{bag}(v, t)$ as follows

$$\min_{v \in V, t \in [n^3]} \Delta\mathsf{bag}(v, t) = \min_{v \in V} \min_{0 \leq t \leq \mathsf{ldr\_time}(v)} \Delta\mathsf{bag}(v, t).$$

We then will distribute the work needed to calculate the right-hand side of the above equality by requiring each vertex to calculate the minimal degree among bags for which it is the leader:

$$\min_{0 \leq t \leq \mathsf{ldr\_time}(v)} \Delta\mathsf{bag}(v, t).$$

Let i be a number in $[\lceil log^2 n \rceil]$. Let L_i (the ith level) be the set of vertices $v \in V$ with low depth decomposition label $\ell(v) = i$, and $L_{\leq i}$ be that with label $\ell(v) \leq i$ (for convenience we assume that $L_{\leq 0} = \emptyset$). Let T^i be the tree T with $L_{\leq i-1}$ removed. The following observation, derived from the fact that a bag is a connected subgraph of T and the leader has lowest value $\ell(\cdot)$, relates bag location to the topology of the low depth decomposition.

Observation 199. *For every* $i \in [\lceil log^2 n \rceil]$*,* $v \in L_i$*, and* $0 \leq t \leq \mathsf{ldr\_time}(v)$*, the set* $\mathsf{bag}(v, t)$ *belongs to a single connected component of graph* T^i*. For any two* $u, v \in L_i$*, sets* $\mathsf{bag}(u, \mathsf{ldr\_time}(u))$ *and* $\mathsf{bag}(v, \mathsf{ldr\_time}(v))$ *belong to different components of graph* T^i.

Recall, that we wanted to calculate the value

$$\min_{0 \le t \le \mathsf{ldr\_time}(v)} \Delta\mathsf{bag}(v, t)$$

for every $v \in V$, which we rewrote as

$$\min_{v \in V} \min_{0 \le t \le \mathsf{ldr\_time}(v)} \Delta\mathsf{bag}(v, t).$$

Grouping by vertices in the same layers, we get

$$\begin{aligned} &\min_{v \in V} \min_{0 \le t \le \mathsf{ldr\_time}(v)} \Delta\mathsf{bag}(v, t) \\ &\qquad = \min_{i \in [\lceil log^2 n \rceil]} \min_{v \in L_i} \min_{0 \le t \le \mathsf{ldr\_time}(v)} \Delta\mathsf{bag}(v, t). \end{aligned}$$

By Observations 199, we can hope that computing the value

$$\min_{v \in L_i} \min_{0 \le t \le \mathsf{ldr\_time}(v)} \Delta\mathsf{bag}(v, t),$$

can be done in parallel without exceeding global memory limit of $O(m \log^2 n)$, since for different $v \in L_i$, their bags up to time $\mathsf{ldr\_time}(v)$ belong to different components of T^i, thus we might avoid redundant work. The details of computing this value are presented in the next section. Let us now formalize the progress so far.

Lemma 200. *Given a tree T and a graph $G = (V, E, w : E \to [n^3])$ as an input, calculating the*

value $\min_{v\in V, t\in[n^3]} \Delta\mathsf{bag}(v,t)$ *can be reduced to* $O(\log^2 n)$ *instances of calculating values*

$$\min_{v\in L_i} \min_{0\le t\le \mathsf{ldr\_time}(v)} \Delta\mathsf{bag}(v,t),$$

for $i \in [\lceil log^2 n\rceil]$. *The reduction can be implemented in AMPC with* $O(1/\epsilon)$ *rounds,* $O((n+m)\log^2 n)$ *total space, and* $O(n^\epsilon)$ *local memory.*

Proof. The correctness follows from the above discussion. For the implementation, the generalized low depth decomposition of T can be determined in $O(1/\epsilon)$ rounds with $O(n\log^2 n)$ total space by Lemma 179. Consider now $O(\log^2)$ tuples of format (T,ℓ,E,L_i). Preparing them requires $O((n+m)\log^2 n)$ total space and the above discussion shows that the value

$$\min_{v\in L_i} \min_{0\le t\le \mathsf{ldr\_time}(v)} \Delta\mathsf{bag}(v,t)$$

for every $i \in [\lceil log^2 n\rceil]$ can be computed from the tuple (T,ℓ,E,L_i), thus the lemma follows. □

16.4.3 Resolving the problem for vertices on the same level.

Following Lemma 200, we fix $i \in [\lceil log^2 n\rceil]$ and set L_i. We calculate:

$$\min_{v\in L_i} \min_{0\le t\le \mathsf{ldr\_time}(v)} \Delta\mathsf{bag}(v,t).$$

In this approach, we will frequently query the minimum value over a path in a tree, thus the following result is helpful.

Theorem 201 (Behnezhad et al. [Behnezhad et al., 2019e])**.** *Consider a rooted, weighted tree*

T, the heavy-light decomposition of this tree together with an RMQ data structure that supports queries on heavy paths can be computed in $O(1/\epsilon)$ AMPC rounds using $O(n^\epsilon)$ local memory and $O(n \log n)$ total space. If the aforementioned data structures are precomputed, then obtaining a minimum value on a path of a tree can be calculated with $O(\log n)$ queries to global memory.

We will also make use of the following theorem.

Theorem 202 (Behnezhad et al. [Behnezhad et al., 2019d])**.** *For a given sequence of integer numbers S of length n, computing the minimum prefix sum over all prefix sums can be done in $O(1/\epsilon)$ AMPC rounds using $O(n^\epsilon)$ local memory and $O(n \log n)$ total space.*

Finally, we show that the construction of the low depth decomposition provided in Section 16.3 gives easy access to edges that connect vertices of higher labels with vertices of smaller labels.

Lemma 203. *For any connected component C^i in T^i, there are at most 2 tree edges between C^i and $V \setminus T^i$ according to the low depth decomposition ℓ given in Lemma 193. Moreover, both edges can be calculated in $O(1/\epsilon)$ AMPC rounds with $O(n^\epsilon)$ memory per machine and $O(n \log^2 n)$ total memory.*

Let us now turn to the proper part of this subsection. First, we show how to compute values $\mathsf{ldr\_time}(v)$ for all $v \in L_i$.

Lemma 204. *Given a tuple (T, ℓ, E, L_i) for tree T, low depth decomposition ℓ, set of weighted edges E, and levels L_i for some $i \in [\lceil \log^2 n \rceil]$, there exists an AMPC algorithm that calculates the value $\mathsf{ldr\_time}(v)$ for every $v \in L_i$, in $O(1/\epsilon)$ rounds using $O(n^\epsilon)$ local memory and $O((n + m) \log^2 n)$ global memory.*

Proof. Consider vertex $v \in L_i$. Vertex v ceases to be the leader of a bag at the first time t when its bag is contracted with another bag containing at least one vertex of the set $L_{\leq i-1}$. According to the tree contraction process, time t is equal to the largest weight of tree edges between v's connected component in graph T^i and the set of vertices $L_{\leq i-1}$. By Lemma 203, these edges can be extracted with at most $O(\log^2 n)$ queries to the low depth decomposition structure. We then simply find the minimum. Thus, all values $\mathsf{ldr\_time}(v)$ for vertices from L_i can be computed in constant number of rounds assumed the conditions stated in the lemma. □

We can assume that values $\mathsf{ldr\_time}(v) \in L_i$ are known. We would like to efficiently compute

$$\min_{0 \leq t \leq \mathsf{ldr\_time}(v)} \Delta\mathsf{bag}(v, t),$$

for each $v \in L_i$. For this, we make the following observation.

Lemma 205. *Consider an edge $(x, y) =: e \in E$ and a vertex $v \in L_i$. All possible values $0 \leq t' \leq \mathsf{ldr\_time}(v)$ at which e belongs to set $\mathsf{nbr\_bag}(v, t')$ form a consecutive (possible empty) interval of integers $[a_e, b_e] \subseteq [0, \ldots, \mathsf{ldr\_time}(v)]$, called also a **time interval** with respect to v.*

Proof. The lemma follows immediately from the fact that $\mathsf{bag}(v, 0) \subseteq \mathsf{bag}(v, 1) \subseteq \ldots \subseteq \mathsf{bag}(v, n^3)$. □

Additionally, the following observation shows, given edge time intervals, how to derive $\min_{0 \leq t \leq \mathsf{ldr\_time}(v)} \Delta\mathsf{bag}(v, t)$ and clarifies the purpose of time intervals.

Observation 206. *Fix a vertex $v \in V$ and consider time intervals $[a_e, b_e]$ with respect to v, for*

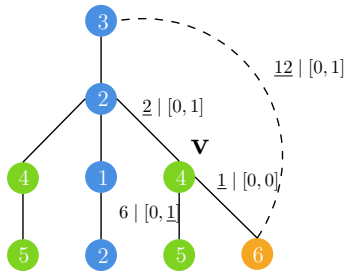


Figure 16.3: A sample structure of an MST tree. Firm edges are *tree* edges, while dotted are *non-tree* edges. The number inside vertices denote their levels. Different colors symbolize different binarized paths. The numbers underlined are times of contraction of corresponding edges. Next to these number the time intervals of these edges with respect to vertex v are given. Since $\mathsf{ldr\_time}(v) = 2$, thus all these intervals are contained in $[0, 2]$.

all $e \in E$. Denote the set of all intervals containing value x by I_x. Then, computing the value

$$\min_{0 \le t \le \mathsf{ldr\_time}(v)} \Delta\mathsf{bag}(v, t),$$

is equivalent to computing the minimum over all values $|I_x|$ for x in the range $[0, \mathsf{ldr\_time}(v)]$.

Since this task is 'linear', it can be computed efficiently in AMPC. We now discuss how to compute the intervals for all edges in E.

Lemma 207. *Given a tuple (T, ℓ, E, L_i), there exists an AMPC algorithm that for every vertex $v \in L_i$ and every edge $e \in E$ calculates the maximal, non-empty time interval $[e_a, e_b] \subseteq [0, \ldots, \mathsf{ldr\_time}(v)]$ of e with respect to v. The algorithm works in $O(1/\epsilon)$ rounds, uses $O(n^\epsilon)$ local memory and $O((n+m)\log^2 n)$ global memory.*

Proof. The algorithm starts by removing vertices $L_{\le i-1}$ with all edges adjacent to them from tree T which gives us T^i. Given decomposition ℓ, this can be done in $O(1)$ rounds. By definition 178, vertices $L_i = \{v_1, \ldots, v_q\}$ belong to different trees. Next, the algorithm roots these trees that contain vertices from L_i in $v_1, \ldots, v_q$ and calculates heavy-light decompositions of each tree

together with an RMQ structure on heavy paths. By Theorem 201, this can be done in $O(1/\epsilon)$ within our memory constraints.

Let us now fix an edge $(x, y) =: e \in E$. Importantly, we consider here all edges of the graph G, not only tree edges E_T. Let $r_x \in \{\perp, v_1, \ldots, v_q\}$ be the root of this tree in T^i to which the vertex x belongs. If the vertex x does not belong to any tree, that is $x \in L_{\leq i-1}$ since these are the vertices that have been removed, we write $r_x = \perp$. Let $\mathsf{mw}(x)$ be the minimum weight over edges of path that connects vertex x with vertex r_x in graph T^i. Observe, that unless $r_x = \perp$ this value is well defined as T^i is a collection of tree and there is exactly one path connecting is x and r_x in this graph. We extend the above definitions on y in the natural way.

By Theorem 201, computing $r_x, r_y, \mathsf{mw}(x), \mathsf{mw}(y)$ takes $O(\log n)$ queries to the memory for a single edge. Therefore, we can compute these values for all edges $e \in E$ in $O(1)$ round under the conditions assumed in this lemma.

Observe that edge $e = (x, y)$ can have non-empty time intervals only with vertices r_x and r_y. Any other vertex from L_i belongs to a different connected component in graph T_i and therefore its bag cannot contain x nor y while the vertex is the leader of its bag. Thus, all that is left to show is how $\mathsf{mw}(x)$ and $\mathsf{mw}(y)$ can help determine the time intervals in which edge e belongs to $\mathsf{nbr\_bag}(r_x)$ and $\mathsf{nbr\_bag}(r_y)$. We consider the following cases.

Case 1. $r_x = \perp, r_y = \perp$. In this case, edge (x, y) has no effect on degrees of bags of vertices r_x and r_y at any time. The algorithm skips such edges.

Case 2. $r_x = \perp, r_y \neq \perp$, (or symmetrically $r_x \neq \perp, r_y = \perp$). Since T^i is a subset of the minimum spanning tree T, thus the first time when vertex x belongs to r_x's bag is the time $\mathsf{mw}(x)$. Now, y starts to belong to r_x's bag either at the time being equal to the maximal weight on the path between r_x and y. Observer however, that this path has to contain vertices that does not belong

to T^i and therefore the maximal weight has to be greater than $\mathsf{ldr\_time}(r_x)$. What follows the correct interval in this case is:

$$[\mathsf{mw}(x), \mathsf{ldr\_time}(r_x)],$$

or an empty interval if $\mathsf{mw}(x) > \mathsf{ldr\_time}(r_x)$.

Case 3. $r_x \neq \perp, r_y \neq \perp$. We distinguish two sub-cases:

Subcase a) $r_x \neq r_y$. Since the path between r_x and r_y does not belong to T^i we can proceed analogously to the *Case* 2.. The correct interval for vertex x is

$$[\mathsf{mw}(x), \mathsf{ldr\_time}(r_x)],$$

or an empty interval if $\mathsf{mw}(x) > \mathsf{ldr\_time}(r_x)$, while for vertex y it is

$$[\mathsf{mw}(y), \mathsf{ldr\_time}(r_y)],$$

or an empty interval if $\mathsf{mw}(y) > \mathsf{ldr\_time}(r_y)$

Subcase b) $r_x = r_y$. Since T^i is a subgraph of the minimum spanning tree T, we have that $\min(\mathsf{mw}(x), \mathsf{mw}(y))$ is the first time when at least one of x and y belongs to r_x's bag, while the first time when both belong to r_x's bag is $\max(\mathsf{mw}(x), \mathsf{mw}(y))$. Thus, the proper time interval for this edge:

$$[\min(\mathsf{mw}(x), \mathsf{mw}(y)), \max(\mathsf{mw}(x), \mathsf{mw}(y))]$$
$$\cap\, [1, \ldots, \mathsf{ldr\_time}(r_x)]$$

We obtain that for every edge $e \in E$ all non-empty time intervals in which this edge belong to nbr_bag of some vertex v can be computed in $O(\log(n))$ queries to the memory. Therefore, computing these values for all edges can be done in constant number of rounds assumed $O(m \log n)$ total memory. □

Implementing Observation 206 is purely technical.

Lemma 208. *There exist an AMPC algorithm that given a set of integer intervals* $I = \{[p_1, k_1], \ldots, [p_n, k_n]\}$, $\forall_{i \in [n]} [p_i, k_i] \subseteq [0, R]$ *finds the minimal number of intersecting intervals in* $O(1/\epsilon)$ *rounds using* $O(n^\epsilon)$ *local memory and* $O(n \log^2 n)$ *total memory.*

Proof. First, the algorithm sorts the set $\{p_1, k_1, \ldots, p_n, k_n\}$ of all endpoints of these intervals in non-increasing order (ties are resolved with priority for endpoints p_i) obtaining a sequence S. Consider assigning to every endpoint $p_i, i \in [n]$ from sequence S value $+1$ and to every endpoint $k_i, i \in [n]$ value -1. This operation leads to a sequence S' of pairs of format (endpoint, value). Finally, let S'' be a sequence constructed from S' in which all consecutive pairs that have the same first coordinate are compressed to a single pair in which the first coordinate is preserved and the second is the sum of second coordinates of contracted pairs. It can be observed that finding the minimal prefix sum of sequence made from second coordinates of pairs in S'' is equivalent to the minimal number of intersecting intervals. The construction of sequence S'' requires only sorting and contracting consecutive pairs which can be implemented in $O(1/\epsilon)$ rounds in AMPC with the memory constrains stated in the lemma. To find the minimal prefix sum we use Theorem 202 which completes the proof. □

The above discussion is summarized in the following lemma.

Lemma 209. *There exists an AMPC algorithm that given a tuple* (T, ℓ, E, L_i) *calculates the value*

$$\min_{v \in L_i} \min_{0 \leq t \leq \mathsf{ldr\_time}(v)} \Delta\mathsf{bag}(v, t)$$

in $O(1/\epsilon)$ *rounds using* $O(n^\epsilon)$ *local memory and* $O((n+m)\log^2 n)$ *total memory.*

Proof. Using Lemma 204 we are able to calculate value $\mathsf{ldr\_time}$ for every $v \in L_i$ in constant number of rounds. By Lemma 207 we can calculate time all non-empty time intervals for every $e \in E$ and every $v \in L_i$. This requires $O(m \log^2 n)$ total memory. Each time interval $[a, b]$ can be assigned a vertex v with respect to whom it was calculated. Then, we group time intervals with respect to vertices from L_i they were calculated. This can be done in a single round with $O(m \log^2 n)$ global memory since there are only $O(m)$ non-empty time intervals. Finally, Lemma 208 guarantees that we can compute, for every $v \in L_i$, the minimum number of intersecting intervals in $O(1/\epsilon)$ rounds with total memory proportional to the number of these intervals. Therefore, assumed $O(m \log^2 n)$ global memory we can extend the last computation to a parallel computation for $v \in L_i$ while preserving the round complexity. By Observation 206 this is equivalent to calculating

$$\min_{0 \leq t \leq \mathsf{ldr\_time}(v)} \Delta\mathsf{bag}(v, t),$$

for every $v \in L_i$. Since the minimum of the above values over $v \in L_i$ can be computed in a single round, the lemma is proven. □

16.4.4 The final algorithm.

We are now able to prove Theorem 194 and present the final algorithm, SmallestSingletonCut, that calculates the smallest singleton cut that appears in the contraction process of G. The pseudcode can be found in Figure 24, while the proof of correctness is below.

Algorithm 24 SmallestSingletonCut

Input Graph $G = (V, E, w : V \to [n^3])$.
Output Size of the smallest singleton cut.

1: Compute the minimum spanning tree T of G
2: Compute the low depth decomposition D_T of T
3: Prepare $O(\log^2 n)$ tuples $(T, D_T, E, L_i), i \in [\lceil log^2 n \rceil]$
4: **for** each tuple (T, D_T, E, L_i) **do**
5: Compute: $\mathsf{lc}_i \leftarrow \min_{v \in L_i} \min_{0 \le t \le \mathsf{ldr\_time}(v)} \Delta\mathsf{bag}(v, t)$
6: **end for**
7: Return $\min(\mathsf{lc}_1, \ldots, \mathsf{lc}_{[\lceil log^2 n \rceil]})$

Proof of Theorem 194. The correctness follows from Observation 196 and Lemmas 200 and 209. Also the implementation details of lines $3-7$ are discussed in the these two lemmas. To calculate minimum spanning tree in line 1 we use Lemma 180 while the implementation of the low depth decomposition from the next line is given by Lemma 179. □

Bibliography

[Acar et al., 2004] Acar, U. A., Blelloch, G. E., Harper, R., Vittes, J. L., and Woo, S. L. M. (2004). Dynamizing static algorithms, with applications to dynamic trees and history independence. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete*, pages 531–540.

[Ahanchi et al., 2023] Ahanchi, A., Andoni, A., Hajiaghayi, M., Knittel, M., and Zhong, P. (2023). Massively parallel tree embeddings for high dimensional spaces. In Agrawal, K. and Shun, J., editors, *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2023, Orlando, FL, USA, June 17-19, 2023*, pages 77–88. ACM.

[Ahmadi et al., 2020] Ahmadi, S., Galhotra, S., Saha, B., and Schwartz, R. (2020). Fair correlation clustering. *arXiv:2002.03508*.

[Ahmadian et al., 2020a] Ahmadian, S., Chatziafratis, V., Epasto, A., Lee, E., Mahdian, M., Makarychev, K., and Yaroslavtsev, G. (2020a). Bisect and conquer: Hierarchical clustering via max-uncut bisection. In *AISTATS*.

[Ahmadian et al., 2020b] Ahmadian, S., Epasto, A., Knittel, M., Kumar, R., Mahdian, M., Moseley, B., Pham, P., Vassilvitskii, S., and Wang, Y. (2020b). Fair hierarchical clustering. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H., editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

[Ahmadian et al., 2019] Ahmadian, S., Epasto, A., Kumar, R., and Mahdian, M. (2019). Clustering without over-representation. In *KDD*, pages 267–275.

[Ahmadian et al., 2020c] Ahmadian, S., Epasto, A., Kumar, R., and Mahdian, M. (2020c). Fair correlation clustering. In *AISTATS*.

[Ahn and Guha, 2015a] Ahn, K. J. and Guha, S. (2015a). Access to Data and Number of Iterations: Dual Primal Algorithms for Maximum Matching under Resource Constraints. In *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2015, Portland, OR, USA, June 13-15, 2015*, pages 202–211.

[Ahn and Guha, 2015b] Ahn, K. J. and Guha, S. (2015b). Access to data and number of iterations: Dual primal algorithms for maximum matching under resource constraints. In *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms*, pages 202–211.

[Ailon and Chazelle, 2006] Ailon, N. and Chazelle, B. (2006). Approximate nearest neighbors and the fast johnson-lindenstrauss transform. In *Proceedings of the 38th Annual ACM Symposium on Theory of Computing*, pages 557–563.

[Alon et al., 2020] Alon, N., Azar, Y., and Vainstein, D. (2020). Hierarchical clustering: a 0.585 revenue approximation. In *COLT*.

[Alon et al., 1986] Alon, N., Babai, L., and Itai, A. (1986). A Fast and Simple Randomized Parallel Algorithm for the Maximal Independent Set Problem. *J. Algorithms*, 7(4):567–583.

[Alon et al., 1995] Alon, N., Karp, R. M., Peleg, D., and West, D. B. (1995). A graph-theoretic game and its application to the k-server problem. *SIAM J. Comput.*, pages 78–100.

[Amgoth and Jana, 2014] Amgoth, T. and Jana, P. K. (2014). Energy efficient and load balanced clustering algorithms for wireless sensor networks. *IJICT*, 6(3/4):272–291.

[Andoni, 2009] Andoni, A. (2009). *Nearest Neighbor Search: the Old, the New, and the Impossible*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA.

[Andoni et al., 2014] Andoni, A., Nikolov, A., Onak, K., and Yaroslavtsev, G. (2014). Parallel algorithms for geometric graph problems. In *Symposium on Theory of Computing*, pages 574–583. ACM.

[Andoni et al., 2018] Andoni, A., Song, Z., Stein, C., Wang, Z., and Zhong, P. (2018). Parallel graph connectivity in log diameter rounds. In *59th IEEE Annual Symposium on Foundations of Computer Science*, pages 674–685. IEEE Computer Society.

[Andoni et al., 2019] Andoni, A., Stein, C., and Zhong, P. (2019). Log diameter rounds algorithms for 2-vertex and 2-edge connectivity. In *46th International Colloquium on Automata, Languages, and Programming*, pages 14:1–14:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[Andoni et al., 2020] Andoni, A., Stein, C., and Zhong, P. (2020). Parallel approximate undirected shortest paths via low hop emulators. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 322–335.

[Angwin et al., 2016] Angwin, J., Larson, J., and Kirchner, L. (2016). Machine bias. *ProPublica*.

[Arifin and Asano, 2006] Arifin, A. Z. and Asano, A. (2006). Image segmentation by histogram thresholding using hierarchical cluster analysis. *Pattern Recognit. Lett.*, 27(13):1515–1521.

[Arjomandi, 1982] Arjomandi, E. (1982). An efficient algorithm for colouring the edges of a graph with Δ+ 1 colours. *INFOR: Information Systems and Operational Research*, 20(2):82–101.

[Arora, 1997] Arora, S. (1997). Nearly linear time approximation schemes for euclidean TSP and other geometric problems. In *38th Annual Symposium on Foundations of Computer Science*, pages 554–563.

[Assadi et al., 2019a] Assadi, S., Bateni, M., Bernstein, A., Mirrokni, V. S., and Stein, C. (2019a). Coresets meet EDCS: algorithms for matching and vertex cover on massive graphs. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete*, pages 1616–1635.

[Assadi et al., 2019b] Assadi, S., Chen, Y., and Khanna, S. (2019b). Sublinear algorithms for $\delta + 1$ vertex coloring. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete*, pages 767–786.

[Assadi et al., 2019c] Assadi, S., Sun, X., and Weinstein, O. (2019c). Massively parallel algorithms for finding well-connected components in sparse graphs. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed*, pages 461–470.

[Atallah et al., 1989] Atallah, M. J., Kosaraju, S. R., Larmore, L. L., Miller, G. L., and Teng, S. (1989). Constructing trees in parallel. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*.

[Backurs et al., 2019] Backurs, A., Indyk, P., Onak, K., Schieber, B., Vakilian, A., and Wagner, T. (2019). Scalable fair clustering. In *ICML*, pages 405–413.

[Bahmani et al., 2014] Bahmani, B., Goel, A., and Munagala, K. (2014). Efficient primal-dual graph algorithms for mapreduce. In Bonato, A., Graham, F. C., and Pralat, P., editors, *Algorithms and Models for the Web Graph - 11th International Workshop, WAW 2014, Beijing, China, December 17-18, 2014, Proceedings*, volume 8882 of *Lecture Notes in Computer Science*, pages 59–78. Springer.

[Barenboim et al., 2016] Barenboim, L., Elkin, M., Pettie, S., and Schneider, J. (2016). The Locality of Distributed Symmetry Breaking. *J. ACM*, 63(3):20:1–20:45.

[Barocas et al., 2019] Barocas, S., Hardt, M., and Narayanan, A. (2019). *Fairness and Machine Learning*. `www.fairmlbook.org`.

[Bartal, 1996] Bartal, Y. (1996). Probabilistic approximations of metric spaces and its algorithmic applications. In *37th Annual Symposium on Foundations of Computer Science*, pages 184–193.

[Bartal, 1998] Bartal, Y. (1998). On approximating arbitrary metrices by tree metrics. In *Proceedings of the Thirtieth Annual ACM Symposium on the Theory*, pages 161–168.

[Bateni et al., 2017] Bateni, M., Behnezhad, S., Derakhshan, M., Hajiaghayi, M., Kiveris, R., Lattanzi, S., and Mirrokni, V. S. (2017). Affinity clustering: Hierarchical clustering at scale. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems*, pages 6864–6874.

[Bateni et al., 2018a] Bateni, M., Behnezhad, S., Derakhshan, M., Hajiaghayi, M., and Mirrokni, V. S. (2018a). Brief announcement: Mapreduce algorithms for massive trees. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*, volume 107 of *LIPIcs*, pages 162:1–162:4. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[Bateni et al., 2018b] Bateni, M., Behnezhad, S., Derakhshan, M., Hajiaghayi, M., and Mirrokni, V. S. (2018b). Massively parallel dynamic programming on trees. *CoRR*.

[Beame et al., 2017] Beame, P., Koutris, P., and Suciu, D. (2017). Communication steps for parallel query processing. *Journal of the ACM (JACM)*, 64(6):1–58.

[Behnezhad, 2021] Behnezhad, S. (2021). Time-optimal sublinear algorithms for matching and vertex cover. *arXiv preprint arXiv:2106.02942*.

[Behnezhad et al., 2019a] Behnezhad, S., Brandt, S., Derakhshan, M., Fischer, M., Hajiaghayi, M., Karp, R. M., and Uitto, J. (2019a). Massively parallel computation of matching and mis in sparse graphs. In *ACM SIGACT*, pages 481–490.

[Behnezhad et al., 2018a] Behnezhad, S., Derakhshan, M., and Hajiaghayi, M. (2018a). Brief Announcement: Semi-MapReduce Meets Congested Clique. *CoRR*, abs/1802.10297.

[Behnezhad et al., 2018b] Behnezhad, S., Derakhshan, M., Hajiaghayi, M., and Karp, R. M. (2018b). Massively parallel symmetry breaking on sparse graphs: MIS and maximal matching. *CoRR*, abs/1807.06701.

[Behnezhad et al., 2019b] Behnezhad, S., Derakhshan, M., Hajiaghayi, M., Knittel, M., and Saleh, H. (2019b). Streaming and massively parallel algorithms for edge coloring. In *27th Annual European Symposium on Algorithms*, pages 15:1–15:14.

[Behnezhad et al., 2019c] Behnezhad, S., Dhulipala, L., Esfandiari, H., Łacki, J., Mirrokni, V., and Schudy, W. (2019c). Massively parallel computation via remote memory access. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 59–68.

[Behnezhad et al., 2019d] Behnezhad, S., Dhulipala, L., Esfandiari, H., Lacki, J., and Mirrokni, V. S. (2019d). Near-optimal massively parallel graph connectivity. In *60th IEEE Annual Symposium on Foundations of Computer Science*, pages 1615–1636.

[Behnezhad et al., 2020] Behnezhad, S., Dhulipala, L., Esfandiari, H., Lacki, J., Mirrokni, V. S., and Schudy, W. (2020). Parallel graph algorithms in constant adaptive rounds: Theory meets practice. *Proc. VLDB Endow.*, pages 3588–3602.

[Behnezhad et al., 2019e] Behnezhad, S., Hajiaghayi, M., and Harris, D. G. (2019e). Exponentially faster massively parallel maximal matching. In *60th IEEE Annual Symposium on Foundations of Computer Science*, pages 1637–1649.

[Ben-Porat et al., 2021] Ben-Porat, O., Sandomirskiy, F., and Tennenholtz, M. (2021). Protecting the protected group: Circumventing harmful fairness. In *Thirty-Fifth AAAI Conference on Artificial Intelligence*, pages 5176–5184. AAAI Press.

[Bera et al., 2019] Bera, S., Chakrabarty, D., Flores, N., and Negahbani, M. (2019). Fair algorithms for clustering. In *NeurIPS*, pages 4955–4966.

[Bercea et al., 2019] Bercea, I. O., Groß, M., Khuller, S., Kumar, A., Rösner, C., Schmidt, D. R., and Schmidt, M. (2019). On the cost of essentially fair clusterings. In *APPROX-RANDOM*, pages 18:1–18:22.

[Blelloch et al., 2012] Blelloch, G. E., Gupta, A., and Tangwongsan, K. (2012). Parallel probabilistic tree embeddings, k-median, and buy-at-bulk network design. In *24th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 205–213.

[Bodlaender et al., 2016] Bodlaender, H. L., Drange, P. G., Dregi, M. S., Fomin, F. V., Lokshtanov, D., and Pilipczuk, M. (2016). A $c^k n$ 5-approximation algorithm for treewidth. *SIAM J. Comput.*, pages 317–378.

[Bodlaender and Hagerup, 1995] Bodlaender, H. L. and Hagerup, T. (1995). Parallel algorithms with optimal speedup for bounded treewidth. In Fülöp, Z. and Gécseg, F., editors, *Automata, Languages and Programming, 22nd International Colloquium, ICALP95, Szeged, Hungary, July 10-14, 1995, Proceedings*, volume 944 of *Lecture Notes in Computer Science*, pages 268–279. Springer.

[Bogen and Rieke, 2018] Bogen, M. and Rieke, A. (2018). Help wanted: An examination of hiring algorithms, equity, and bias. Technical report, Upturn.

[Boroujeni et al., 2018] Boroujeni, M., Ehsani, S., Ghodsi, M., Hajiaghayi, M. T., and Seddighin, S. (2018). Approximating edit distance in truly subquadratic time: Quantum and mapreduce. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete*, pages 1170–1189.

[Borůvka, 1926] Borůvka, O. (1926). *O jistém problému minimálním*. Práce Moravské přírodovědecké společnosti. Mor. přírodovědecká společnost.

[Brandt et al., 2018] Brandt, S., Fischer, M., and Uitto, J. (2018). Matching and MIS for uniformly sparse graphs in the low-memory MPC model. *CoRR*, abs/1807.05374.

[Brubach et al., 2020] Brubach, B., Chakrabarti, D., Dickerson, J. P., Khuller, S., Srinivasan, A., and Tsepenekas, L. (2020). A pairwise fair and community-preserving approach to k-center clustering. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 1178–1189. PMLR.

[Cai et al., 2004] Cai, D., He, X., Li, Z., Ma, W., and Wen, J. (2004). Hierarchical clustering of WWW image search results using visual, textual and link information. In *Proceedings of the 12th ACM International Conference on Multimedia*, pages 952–959. ACM.

[Celis et al., 2018] Celis, L. E., Huang, L., and Vishnoi, N. K. (2018). Multiwinner voting with fairness constraints. In *IJCAI*, pages 144–151.

[Chakrabarti et al., 2021] Chakrabarti, D., Dickerson, J. P., Esmaeili, S. A., Srinivasan, A., and Tsepenekas, L. (2021). A new notion of individually fair clustering: α-equitable k-center. *CoRR*, abs/2106.05423.

[Chambers et al., 2010] Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R. R., Bradshaw, R., and Weizenbaum, N. (2010). Flumejava: easy, efficient data-parallel pipelines. In mand Alexander Aiken, B. G. Z., editor, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 363–375.

[Chan et al., 2005] Chan, H. T., Gupta, A., Maggs, B. M., and Zhou, S. (2005). On hierarchical routing in doubling metrics. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, Vancouver, British Columbia, Canada, January 23-25, 2005*, pages 762–771. SIAM.

[Chang et al., 2018] Chang, Y.-J., Fischer, M., Ghaffari, M., Uitto, J., and Zheng, Y. (2018). The Complexity of $(\Delta + 1)$ Coloring in Congested Clique, Massively Parallel Computation, and Centralized Local Computation. *CoRR*, abs/1808.08419.

[Charikar and Chatziafratis, 2017] Charikar, M. and Chatziafratis, V. (2017). Approximate hierarchical clustering via sparsest cut and spreading metrics. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 841–854.

[Charikar et al., 2019a] Charikar, M., Chatziafratis, V., and Niazadeh, R. (2019a). Hierarchical clustering better than average-linkage. In *SODA*, pages 2291–2304.

[Charikar et al., 2019b] Charikar, M., Chatziafratis, V., Niazadeh, R., and Yaroslavtsev, G. (2019b). Hierarchical clustering for euclidean data. In *AISTATS*, pages 2721–2730.

[Charikar et al., 2004] Charikar, M., Chekuri, C., Feder, T., and Motwani, R. (2004). Incremental clustering and dynamic information retrieval. *SIAM J. Comput.*, 33(6):1417–1440.

[Charikar et al., 1998] Charikar, M., Chekuri, C., Goel, A., Guha, S., and Plotkin, S. A. (1998). Approximating a finite metric by a small number of tree metrics. In *39th Annual Symposium on Foundations of Computer Science*, pages 379–388.

[Charikar et al., 2020] Charikar, M., Ma, W., and Tan, L. (2020). Unconditional lower bounds for adaptive massively parallel computation. In *SPAA '20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 141–151.

[Chatziafratis et al., 2018] Chatziafratis, V., Niazadeh, R., and Charikar, M. (2018). Hierarchical clustering with structural constraints. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 773–782. PMLR.

[Chen et al., 2021] Chen, J., Zhang, S., He, X., Lin, Q., Zhang, H., Hao, D., Kang, Y., Gao, F., Xu, Z., Dang, Y., and Zhang, D. (2021). How incidental are the incidents? characterizing and prioritizing incidents for large-scale online service systems. In *ASE '20: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. Association for Computing Machinery.

[Chen et al., 2019] Chen, X., Fain, B., Lyu, C., and Munagala, K. (2019). Proportionally fair clustering. In *ICML*, pages 1032–1041.

[Chen et al., 2020a] Chen, X., Jayaram, R., Levi, A., and Waingarten, E. (2020a). An improved analysis of the quadtree for high dimensional emd. https://rajeshjayaram.com/EarthMoverCJLW.pdf.

[Chen et al., 2020b] Chen, Y., Yang, X., Dong, H., He, X., Zhang, H., Lin, Q., Chen, J., Zhao, P., Kang, Y., Gao, F., Xu, Z., and Zhang, D. (2020b). Identifying linked incidents in large-scale online service systems. In *ESEC/FSE 2020: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery.

[Chhabra and Mohapatra, 2022] Chhabra, A. and Mohapatra, P. (2022). Fair algorithms for hierarchical agglomerative clustering. In Wani, M. A., Kantardzic, M. M., Palade, V., Neagu, D., Yang, L., and Chan, K. Y., editors, *21st IEEE International Conference on Machine Learning and Applications, ICMLA 2022, Nassau, Bahamas, December 12-14, 2022*, pages 206–211. IEEE.

[Chierichetti et al., 2017] Chierichetti, F., Kumar, R., Lattanzi, S., and Vassilvitskii, S. (2017). Fair clustering through fairlets. In *NIPS*, pages 5029–5037.

[Chierichetti et al., 2019] Chierichetti, F., Kumar, R., Lattanzi, S., and Vassilvitskii, S. (2019). Matroids, matchings, and fairness. In *AISTATS*, pages 2212–2220.

[Chiplunkar et al., 2020] Chiplunkar, A., Kale, S., and Ramamoorthy, S. N. (2020). How to solve fair k-center in massive data models. In *ICML*.

[Chitnis et al., 2016] Chitnis, R., Cormode, G., Esfandiari, H., Hajiaghayi, M., McGregor, A., Monemizadeh, M., and Vorotnikova, S. (2016). Kernelization via sampling with applications to finding matchings and related problems in dynamic graph streams. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 1326–1344.

[Chitnis et al., 2015] Chitnis, R. H., Cormode, G., Hajiaghayi, M. T., and Monemizadeh, M. (2015). Parameterized streaming: Maximal matching and vertex cover. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 1234–1251.

[Cohen-Addad et al., 2017] Cohen-Addad, V., Kanade, V., and Mallmann-Trenn, F. (2017). Hierarchical clustering beyond the worst-case. In *NIPS*, pages 6201–6209.

[Cohen-Addad et al., 2018] Cohen-Addad, V., Kanade, V., Mallmann-Trenn, F., and Mathieu, C. (2018). Hierarchical clustering: Objective functions and algorithms. In *SODA*, pages 378–397.

[Cole and Vishkin, 1988] Cole, R. and Vishkin, U. (1988). The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time. *Algorithmica*, pages 329–346.

[Czumaj et al., 2018] Czumaj, A., Lacki, J., Madry, A., Mitrovic, S., Onak, K., and Sankowski, P. (2018). Round compression for parallel matching algorithms. In Diakonikolas, I., Kempe, D., and Henzinger, M., editors, *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*, pages 471–484. ACM.

[Dasgupta, 2016] Dasgupta, S. (2016). A cost function for similarity-based hierarchical clustering. In *STOC*, pages 118–127.

[Dean and Ghemawat, 2008] Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113.

[Dekel et al., 1986] Dekel, E., Ntafos, S., and Peng, S.-T. (1986). Parallel tree techniques and code optimization. In Makedon, F., Mehlhorn, K., Papatheodorou, T., and Spirakis, P., editors, *VLSI Algorithms and Architectures*, pages 205–216.

[Demetrescu et al., 2006] Demetrescu, C., Finocchi, I., and Ribichini, A. (2006). Trading off space for passes in graph streaming problems. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, Miami, Florida, USA, January 22-26, 2006*, pages 714–723.

[Dengel et al., 2011] Dengel, A., Althoff, T., and Ulges, A. (2011). Balanced clustering for content-based image browsing. In *German Computer Science Society, Informatiktage*.

[Dragojevic et al., 2017] Dragojevic, A., Narayanan, D., and Castro, M. (2017). Rdma reads: To use or not to use? *IEEE Data Eng. Bull.*, 40(1):3–14.

[Dragojević et al., 2014] Dragojević, A., Narayanan, D., Castro, M., and Hodson, O. (2014). Farm: Fast remote memory. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 401–414.

[Dua and Graff, 2017] Dua, D. and Graff, C. (2017). UCI machine learning repository.

[Dubes and Jain, 1980] Dubes, R. and Jain, A. K. (1980). Clustering methodologies in exploratory data analysis. In *Advances in Computers*, volume 19, pages 113–228. Academic Press.

[Dwork et al., 2012] Dwork, C., Hardt, M., Pitassi, T., Reingold, O., and Zemel, R. S. (2012). Fairness through awareness. In Goldwasser, S., editor, *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012*, pages 214–226. ACM.

[Epasto et al., 2021] Epasto, A., Mahdian, M., Mirrokni, V. S., and Zhong, P. (2021). Massively parallel and dynamic algorithms for minimum size clustering. *CoRR*, abs/2106.02685.

[Esmaeili et al., 2021] Esmaeili, S. A., Brubach, B., Srinivasan, A., and Dickerson, J. (2021). Fair clustering under a bounded cost. In *Advances in Neural Information Processing Systems*, pages 14345–14357.

[Esmaeili et al., 2020] Esmaeili, S. A., Brubach, B., Tsepenekas, L., and Dickerson, J. (2020). Probabilistic fair clustering. In *Advances in Neural Information Processing Systems*.

[Fakcharoenphol et al., 2003] Fakcharoenphol, J., Rao, S., and Talwar, K. (2003). A tight bound on approximating arbitrary metrics by tree metrics. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing*, pages 448–455.

[Feige and Krauthgamer, 2000] Feige, U. and Krauthgamer, R. (2000). A polylogarithmic approximation of the minimum bisection. In *FOCS*, pages 105–115.

[Feigenbaum et al., 2004] Feigenbaum, J., Kannan, S., McGregor, A., Suri, S., and Zhang, J. (2004). On graph problems in a semi-streaming model. In *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings*, pages 531–543.

[Ferragina and Gulli, 2005] Ferragina, P. and Gulli, A. (2005). A personalized search engine based on web-snippet hierarchical clustering. In *Proceedings of the 14th international conference on World Wide Web*, pages 801–810. ACM.

[Fraigniaud et al., 2016] Fraigniaud, P., Heinrich, M., and Kosowski, A. (2016). Local conflict coloring. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*, pages 625–634.

[Friedrichs and Lenzen, 2018] Friedrichs, S. and Lenzen, C. (2018). Parallel metric tree embedding based on an algebraic view on moore-bellman-ford. *Journal of the ACM (JACM)*, 65(6):1–55.

[Gabow et al., 1985] Gabow, H. N., Nishizeki, T., Kariv, O., Leven, D., , and Terada, O. (1985). Algorithms for edgecoloring graphs. Technical report, Tohoku University.

[Garey and Johnson, 2002] Garey, M. R. and Johnson, D. S. (2002). *Computers and Intractability*. WH Freeman New York.

[Gazit and Miller, 1987] Gazit, H. and Miller, G. L. (1987). A parallel algorithm for finding a separator in planar graphs. In *28th Annual Symposium on Foundations of Computer Science*, pages 238–248.

[Gazit et al., 1988] Gazit, H., Miller, G. L., and Teng, S. (1988). Optimal tree contraction in an EREW model. In *Concurrent Computations: Algorithms, Architecture and Technology*, pages 139–156.

[Ghaffari et al., 2018] Ghaffari, M., Gouleakis, T., Konrad, C., Mitrovic, S., and Rubinfeld, R. (2018). Improved massively parallel computation algorithms for mis, matching, and vertex cover. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed*, pages 129–138.

[Ghaffari et al., 2020] Ghaffari, M., Grunau, C., and Jin, C. (2020). Improved mpc algorithms for mis, matching, and coloring on trees and beyond. *arXiv preprint arXiv:2002.09610*.

[Ghaffari et al., 2019a] Ghaffari, M., Kuhn, F., and Uitto, J. (2019a). Conditional hardness results for massively parallel computation from distributed lower bounds. In *60th IEEE Annual Symposium on Foundations of Computer Science*, pages 1650–1663.

[Ghaffari et al., 2019b] Ghaffari, M., Lattanzi, S., and Mitrovic, S. (2019b). Improved parallel algorithms for density-based network clustering. In Chaudhuri, K. and Salakhutdinov, R., editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019,*

9-15 June 2019, Long Beach, California, USA, volume 97 of *Proceedings of Machine Learning Research*, pages 2201–2210. PMLR.

[Ghaffari and Nowicki, 2020] Ghaffari, M. and Nowicki, K. (2020). Massively parallel algorithms for minimum cut. In Emek, Y. and Cachin, C., editors, *PODC '20: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, August 3-7, 2020*, pages 119–128. ACM.

[Ghaffari and Uitto, 2019] Ghaffari, M. and Uitto, J. (2019). Sparsifying distributed algorithms with ramifications in massively parallel computation and centralized local computation. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1636–1653. SIAM.

[Gibbons and Rytter, 1989] Gibbons, A. and Rytter, W. (1989). Optimal parallel algorithms for dynamic expression evaluation and context-free recognition. *Inf. Comput.*, page 32–45.

[Glazik et al., 2017] Glazik, C., Schiemann, J., and Srivastav, A. (2017). Finding euler tours in one pass in the w-streaming model with o(n log(n)) RAM. *CoRR*, abs/1710.04091.

[Goldberg et al., 1987] Goldberg, A., Plotkin, S., and Shannon, G. (1987). Parallel Symmetry-breaking in Sparse Graphs. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 315–324, New York, NY, USA. ACM.

[Goldberg and Plotkin, 1987] Goldberg, A. V. and Plotkin, S. A. (1987). Parallel $(\Delta + 1)$-coloring of Constant-degree Graphs. *Inf. Process. Lett.*, 25(4):241–245.

[Gomory and Hu, 1961] Gomory, R. E. and Hu, T. C. (1961). Multi-terminal network flows. *Journal of the Society for Industrial and Applied Mathematics*, 9(4):551–570.

[Goodrich and Kosaraju, 1996] Goodrich, M. T. and Kosaraju, S. R. (1996). Sorting on a parallel pointer machine with applications to set expression evaluation. *J. ACM*, pages 331–361.

[Goodrich et al., 2011] Goodrich, M. T., Sitchinava, N., and Zhang, Q. (2011). Sorting, searching, and simulation in the mapreduce framework. In *International Symposium on Algorithms and Computation*, pages 374–383. Springer.

[Grohe and Verbitsky, 2006] Grohe, M. and Verbitsky, O. (2006). Testing graph isomorphism in parallel by playing a game. In *Automata, Languages and Programming, 33rd International Colloquium*, pages 3–14.

[Gupta et al., 2006] Gupta, A., Hajiaghayi, M. T., and Räcke, H. (2006). Oblivious network design. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, Miami, Florida, USA, January 22-26, 2006*, pages 970–979. ACM Press.

[Gupta et al., 2003] Gupta, A., Krauthgamer, R., and Lee, J. R. (2003). Bounded geometries, fractals, and low-distortion embeddings. In *44th Symposium on Foundations of Computer Science (FOCS 2003), 11-14 October 2003, Cambridge, MA, USA, Proceedings*, pages 534–543. IEEE Computer Society.

[Gupta et al., 2004] Gupta, A., Newman, I., Rabinovich, Y., and Sinclair, A. (2004). Cuts, trees and l_1-embeddings of graphs. *Comb.*, 24(2):233–269.

[Hajiaghayi and Knittel, 2020] Hajiaghayi, M. and Knittel, M. (2020). Matching affinity clustering: Improved hierarchical clustering at scale with guarantees. In *Proceedings of the 19th International Conference on Autonomous Agents*, pages 1864–1866.

[Hajiaghayi et al., 2022a] Hajiaghayi, M., Knittel, M., Olkowski, J., and Saleh, H. (2022a). Adaptive massively parallel algorithms for cut problems. In Agrawal, K. and Lee, I. A., editors, *SPAA '22: 34th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 23–33. ACM.

[Hajiaghayi et al., 2022b] Hajiaghayi, M., Knittel, M., Saleh, H., and Su, H. (2022b). Adaptive massively parallel constant-round tree contraction.

[Hajiaghayi et al., 2021] Hajiaghayi, M., Saleh, H., Seddighin, S., and Sun, X. (2021). String matching with wildcards in the massively parallel computation model. In *SPAA*, pages 275–284.

[Hajiaghayi et al., 2019] Hajiaghayi, M., Seddighin, S., and Sun, X. (2019). Massively parallel approximation algorithms for edit distance and longest common subsequence. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1654–1672. SIAM.

[Harris et al., 2016] Harris, D. G., Schneider, J., and Su, H.-H. (2016). Distributed $(\Delta + 1)$-coloring in sublogarithmic rounds. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 465–478. ACM.

[Harvey et al., 2018] Harvey, N. J., Liaw, C., and Liu, P. (2018). Greedy and Local Ratio Algorithms in the MapReduce Model. In *Proceedings of the 30th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2018*.

[Hassin et al., 1997] Hassin, R., Rubinstein, S., and Tamir, A. (1997). Approximation algorithms for maximum dispersion. *Oper. Res. Lett.*, 21(3):133–137.

[Huang et al., 2019] Huang, L., Jiang, S. H. C., and Vishnoi, N. K. (2019). Coresets for clustering with fairness constraints. In *NeurIPS*, pages 7587–7598.

[Im et al., 2017] Im, S., Moseley, B., and Sun, X. (2017). Efficient massively parallel methods for dynamic programming. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 798–811.

[Jez and Lohrey, 2016] Jez, A. and Lohrey, M. (2016). Approximation of smallest linear tree grammar. *Inf. Comput.*, pages 215–251.

[Jin et al., 2015] Jin, C., Liu, R., Chen, Z., Hendrix, W., Agrawal, A., and Choudhary, A. N. (2015). A scalable hierarchical clustering algorithm using spark. In *First IEEE International Conference on Big Data Computing Service and Applications, BigDataService 2015, Redwood City, CA, USA, March 30 - April 2, 2015*, pages 418–426.

[Jin et al., 2013] Jin, C., Patwary, M. M. A., Hendrix, W., Agrawal, A., Liao, W.-k., and Choudhary, A. (2013). Disc: A distributed single-linkage hierarchical clustering algorithm using mapreduce. *International Workshop on Data Intensive Computing in the Clouds (DataCloud)*.

[Johansson, 1999] Johansson, Ö. (1999). Simple distributed *Delta*+1-coloring of graphs. *Inf. Process. Lett.*, 70(5):229–232.

[Johnson and Lindenstrauss, 1984] Johnson, W. and Lindenstrauss, J. (1984). Extensions of lipschitz maps into a hilbert space. *Contemporary Mathematics*, pages 189–206.

[Jones et al., 2020] Jones, M., Nguyen, T., and Nguyen, H. (2020). Fair k-centers via maximum matching. In *ICML*.

[Jurdzinski and Nowicki, 2018] Jurdzinski, T. and Nowicki, K. (2018). MST in $O(1)$ rounds of congested clique. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 2620–2632.

[Karger, 1993] Karger, D. R. (1993). Global min-cuts in rnc, and other ramifications of a simple min-cut algorithm. In Ramachandran, V., editor, *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas, USA*, pages 21–30. ACM/SIAM.

[Karger and Stein, 1996] Karger, D. R. and Stein, C. (1996). A new approach to the minimum cut problem. *Journal of the ACM (JACM)*, 43(4):601–640.

[Karloff et al., 2010] Karloff, H. J., Suri, S., and Vassilvitskii, S. (2010). A model of computation for mapreduce. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 938–948.

[Karp, 1989] Karp, R. (1989). A 2k-competitive algorithm for the circle. Manuscript.

[Karypis et al., 2000] Karypis, M. S. G., Kumar, V., and Steinbach, M. (2000). A comparison of document clustering techniques. In *TextMining Workshop at KDD*.

[Kearns and Roth, 2020] Kearns, M. and Roth, A. (2020). *The Ethical Algorithm*. Oxford University Press.

[Kiveris et al., 2014] Kiveris, R., Lattanzi, S., Mirrokni, V. S., Rastogi, V., and Vassilvitskii, S. (2014). Connected components in mapreduce and beyond. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 18:1–18:13.

[Kleinberg et al., 2017a] Kleinberg, J., Lakkaraju, H., Leskovec, J., Ludwig, J., and Mullainathan, S. (2017a). Human decisions and machine predictions. *The Quarterly Journal of Economics*, 133(1):237–293.

[Kleinberg et al., 2017b] Kleinberg, J. M., Mullainathan, S., and Raghavan, M. (2017b). Inherent trade-offs in the fair determination of risk scores. In *ITCS*, pages 43:1–43:23.

[Kleindessner et al., 2019a] Kleindessner, M., Awasthi, P., and Morgenstern, J. (2019a). Fair k-center clustering for data summarization. In *ICML*, pages 3448–3457.

[Kleindessner et al., 2019b] Kleindessner, M., Samadi, S., Awasthi, P., and Morgenstern, J. (2019b). Guarantees for spectral clustering with fairness constraints. In *ICML*, pages 3458–3467.

[Knittel et al., 2023a] Knittel, M., Springer, M., Dickerson, J. P., and Hajiaghayi, M. (2023a). Fair, polylog-approximate low-cost hierarchical clustering. In *Annual Conference on Neural Information Processing Systems 2023*.

[Knittel et al., 2023b] Knittel, M., Springer, M., Dickerson, J. P., and Hajiaghayi, M. (2023b). Generalized reductions: Making any hierarchical clustering fair and balanced with low cost. In Krause, A., Brunskill, E., Cho, K., Engelhardt, B., Sabato, S., and Scarlett, J., editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 17218–17242. PMLR.

[Konjevod et al., 2001] Konjevod, G., Ravi, R., and Salman, F. S. (2001). On approximating planar metrics by tree metrics. *Inf. Process. Lett.*, pages 213–219.

[Kou and Lou, 2012] Kou, G. and Lou, C. (2012). Multiple factor hierarchical clustering algorithm for large scale web page and search engine clickstream data. *Ann. Oper. Res.*, 197(1):123–134.

[Kraskov et al., 2003] Kraskov, A., Stögbauer, H., Andrzejak, R. G., and Grassberger, P. (2003). Hierarchical clustering using mutual information. *CoRR*, q-bio.QM/0311037.

[Kuhn and Wattenhofer, 2006] Kuhn, F. and Wattenhofer, R. (2006). On the Complexity of Distributed Graph Coloring. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Principles of Distributed Computing*, PODC '06, pages 7–15, New York, NY, USA. ACM.

[Lacki et al., 2020] Lacki, J., Mitrovic, S., Onak, K., and Sankowski, P. (2020). Walking randomly, massively, and efficiently. In *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory*, pages 364–377.

[Lattanzi et al., 2011] Lattanzi, S., Moseley, B., Suri, S., and Vassilvitskii, S. (2011). Filtering: a method for solving graph problems in mapreduce. In Rajaraman, R. and auf der Heide, F. M., editors, *SPAA 2011: Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Jose, CA, USA, June 4-6, 2011 (Co-located with FCRC 2011)*, pages 85–94. ACM.

[Ledford, 2019] Ledford, H. (2019). Millions of black people affected by racial bias in healthcare algorithms. *Nature*, 574.

[Lee and Naor, 2004] Lee, J. R. and Naor, A. (2004). Embedding the diamond graph in l p and dimension reduction in l 1. *Geometric & Functional Analysis GAFA*, 14(4):745–747.

[Leskovec et al., 2014] Leskovec, J., Rajaraman, A., and Ullman, J. D. (2014). *Mining of Massive Datasets, 2nd Ed.* Cambridge University Press.

[Lin et al., 2006] Lin, G., Nagarajan, C., Rajaraman, R., and Williamson, D. P. (2006). A general approach for incremental approximation and hierarchical clustering. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, Miami, Florida, USA, January 22-26, 2006*, pages 1147–1156.

[Lin et al., 2018] Lin, W., Chen, J., Ranjan, R., Bansal, A., Sankaranarayanan, S., Castillo, C. D., and Chellappa, R. (2018). Proximity-aware hierarchical clustering of unconstrained faces. *Image Vis. Comput.*, 77:33–44.

[Linial, 1992] Linial, N. (1992). Locality in Distributed Graph Algorithms. *SIAM J. Comput.*, 21(1):193–201.

[Luby, 1985] Luby, M. (1985). A Simple Parallel Algorithm for the Maximal Independent Set Problem. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, pages 1–10, New York, NY, USA. ACM.

[Ludwig, 2015] Ludwig, S. A. (2015). Mapreduce-based fuzzy c-means clustering algorithm: implementation and scalability. *Int. J. Machine Learning & Cybernetics*, 6(6):923–934.

[Mann et al., 2008] Mann, C. F., Matula, D. W., and Olinick, E. V. (2008). The use of sparsest cuts to reveal the hierarchical community structure of social networks. *Soc. Networks*, 30(3):223–234.

[Miller and Ramachandran, 1987] Miller, G. L. and Ramachandran, V. (1987). A new graph triconnectivity algorithm and its parallelization. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 335–344.

[Miller et al., 1988] Miller, G. L., Ramachandran, V., and Kaltofen, E. (1988). Efficient parallel evaluation of straight-line code and arithmetic circuits. *SIAM J. Comput.*, pages 687–695.

[Miller and Reif, 1985] Miller, G. L. and Reif, J. H. (1985). Parallel tree contraction and its application. In *26th Annual Symposium on Foundations of Computer Science*, pages 478–489.

[Miller and Reif, 1989] Miller, G. L. and Reif, J. H. (1989). Parallel tree contraction part 1: Fundamentals. *Adv. Comput. Res.*, pages 47–72.

[Miller and Reif, 1991] Miller, G. L. and Reif, J. H. (1991). Parallel tree contraction, part 2: Further applications. *SIAM J. Comput.*, pages 1128–1147.

[Moseley and Wang, 2017] Moseley, B. and Wang, J. (2017). Approximation bounds for hierarchical clustering: Average linkage, bisecting k-means, and local search. In *NIPS*, pages 3094–3103.

[Nanongkai and Scquizzato, 2019] Nanongkai, D. and Scquizzato, M. (2019). Equivalence classes and conditional hardness in massively parallel fcomputations. In *23rd International Conference on Principles of Distributed Systems*, pages 33:1–33:16.

[Naor and Schechtman, 2006] Naor, A. and Schechtman, G. (2006). Planar earthmover is not in l_1. In *47th Annual IEEE Symposium on Foundations of Computer Science*, pages 655–666. IEEE Computer Society.

[Pan et al., 2016] Pan, T., Lo, L., Yeh, C., Li, J., Liu, H., and Hu, M. (2016). Real-time sign language recognition in complex background scene based on a hierarchical clustering classification method. In *IEEE Second International Conference on Multimedia Big Data*, pages 64–67. IEEE Computer Society.

[Panconesi and Srinivasan, 1992] Panconesi, A. and Srinivasan, A. (1992). Improved Distributed Algorithms for Coloring and Network Decomposition Problems. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Theory of Computing*, STOC '92, pages 581–592, New York, NY, USA. ACM.

[Papadopoulos et al., 2015] Papadopoulos, D., Papamanthou, C., Tamassia, R., and Triandopoulos, N. (2015). Practical authenticated pattern matching with optimal proof size. *Proc. VLDB Endow.*, pages 750–761.

[Parter, 2018] Parter, M. (2018). $(\Delta + 1)$ Coloring in the Congested Clique Model. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*, pages 160:1–160:14.

[Parter and Su, 2018] Parter, M. and Su, H. (2018). Randomized $(\Delta+1)$-Coloring in $O(\log^* \Delta)$ Congested Clique Rounds. In *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, pages 39:1–39:18.

[Rabinovich and Raz, 1998] Rabinovich, Y. and Raz, R. (1998). Lower bounds on the distortion of embedding finite metric spaces in graphs. *Discret. Comput. Geom.*, 19(1):79–94.

[Rajaraman and Ullman, 2011] Rajaraman, A. and Ullman, J. D. (2011). *Mining of Massive Datasets*. Cambridge University Press.

[Ramanath et al., 2013] Ramanath, R., Choudhury, M., and Bali, K. (2013). Entailment: An effective metric for comparing and evaluating hierarchical and non-hierarchical annotation schemes. In Dipper, S., Liakata, M., and Pareja-Lora, A., editors, *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse, LAW-ID@ACL 2013, August 8-9, 2013, Sofia, Bulgaria*, pages 42–50. The Association for Computer Linguistics.

[Rand, 1971] Rand, W. M. (1971). Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical Association*, 66(336):846–850.

[Rao, 1999] Rao, S. (1999). Small distortion and volume preserving embeddings for planar and euclidean metrics. In *Proceedings of the Fifteenth Annual Symposium on Computational Geometry*, SCG '99, page 300–306, New York, NY, USA. Association for Computing Machinery.

[Rieke and Bogen, 2018] Rieke, A. and Bogen, M. (2018). Help wanted: An examination of hiring algorithms, equity, and bias. *Upturn*.

[Rösner and Schmidt, 2018] Rösner, C. and Schmidt, M. (2018). Privacy preserving clustering with constraints. In *ICALP*, pages 96:1–96:14.

[Roughgarden et al., 2016] Roughgarden, T., Vassilvitskii, S., and Wang, J. R. (2016). Shuffles and circuits: (on lower bounds for modern parallel computation). In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms*, pages 1–12.

[Roy and Pokutta, 2016] Roy, A. and Pokutta, S. (2016). Hierarchical clustering via spreading metrics. In *NIPS*, pages 2316–2324.

[Saran and Vazirani, 1995] Saran, H. and Vazirani, V. V. (1995). Finding k cuts within twice the optimal. *SIAM J. Comput.*, 24(1):101–108.

[Schmidt et al., 2019] Schmidt, M., Schwiegelshohn, C., and Sohler, C. (2019). Fair coresets and streaming algorithms for fair k-means. In Bampis, E. and Megow, N., editors, *WAOA*, pages 232–251.

[Selvan et al., 2005] Selvan, A., Cole, L., Spackman, L., Naylor, S., and C, W. (2005). Hierarchical cluster analysis to aid diagnostic image data visualization of ms and other medical imaging modalities. In *Molecular Biotechnology*.

[Sleator and Tarjan, 1981] Sleator, D. D. and Tarjan, R. E. (1981). A data structure for dynamic trees. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing, May 11-13, 1981, Milwaukee, Wisconsin, USA*, pages 114–122. ACM.

[Sweeney, 2013] Sweeney, L. (2013). Discrimination in online ad delivery. *ACM Queue*, 11(3):10.

[Vizing, 1964] Vizing, V. G. (1964). On an estimate of the chromatic class of a p-graph. *Discret Analiz*, 3:25–30.

[White, 2009] White, T. (2009). *Hadoop - The Definitive Guide: MapReduce for the Cloud*. O'Reilly.

[Yaroslavtsev and Vadapalli, 2018] Yaroslavtsev, G. and Vadapalli, A. (2018). Massively parallel algorithms and hardness for single-linkage clustering under ℓ_p distances. In *Proceedings of the 35th International Conference on Machine Learning*, pages 5596–5605.

[Zaharia et al., 2010] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*.

[Zaharia et al., 2016] Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S., and Stoica, I. (2016). Apache spark: a unified engine for big data processing. *Commun. ACM*, pages 56–65.